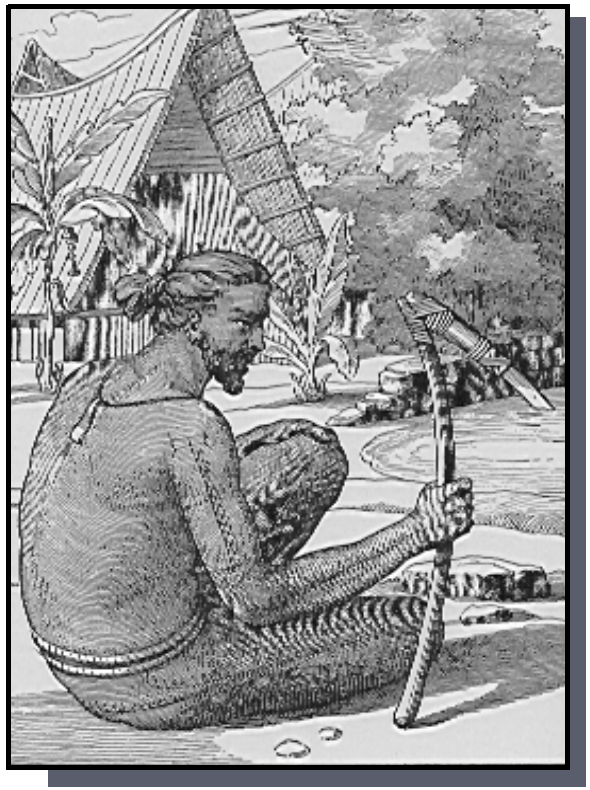
Software Engineering - Wartung und Qualitätssicherung



Not a Java Warrior with his favourite engineering tool!

Prof. Dr. Andy Schürr
Fachgebiet Echzeitsysteme
FB ETiT (Informatik)

Technische Universität Darmstadt,
Merckstr. 25, D-64283 Darmstadt

Andy.Schuerr@es.tu-darmstadt.de

Tel.: 06151 / 16-6940
Raum: S 306 / 313

WWW-Seite der Vorlesung:

http://www.es.tu-darmstadt.de/lehre/se-ii-v/

Bildquelle:

Jules Verne: The Great Explorers of the XIXth Century,
New York: Charles Scribner's Sons (1912)

WWW-Seite der Vorlesung:

http://www.es.tu-darmstadt.de/Lehre/se-ii-v/

Verankerung als Wahlmodul in Prüfungsplänen von Studiengängen:

- ETiT / DT, ...
- (Wirtschafts-)Informatik
- Informationssystemtechnik, Mechatronik, ...

Termine der Lehrveranstaltung:

- **Vorlesung**:
  Montag 10:00 (s.t.) bis 11:30 Uhr in S1|01|A02
  Donnerstag 8:00 (s.t.) bis 8:45 in S1|01|A01 (am 23.10. bis 9:30)
- **Übung ab 30.10.2014**:
  Donnerstag 8:45 bis 9:30 in S1|01|A01

Wissensgebiete der Software-Technik:

Der IEEE Computer Society „Guide to the Software Engineering Body of Knowledge“ (SWEBOK, http://www.swebok.org) zählt folgende Wissensgebiete auf:

1. **Software Requirements**:

   es wird festgelegt, „**was**“ ein Software-System leisten soll (und warum)

2. **Software Design**:

   das „**Wie**“ steht nun im Vordergrund, der Bauplan (Architektur)

3. **Software Construction**:

   gemäß Bauplan wird das Software-System realisiert

4. **Software Testing**:

   Fehler werden **systematisch** gesucht und eliminiert

5. **Software Maintenance**:

   die Pflege und Weiterentwicklung der Software nach Auslieferung

6. **Software Configuration Management**:

   die Verwaltung von Software-Versionen und -Konfigurationen

7. **Software Engineering Management**:

   (Projekt-)Management von Personen, Organisationen, Zeitplänen, …

8. **Software Engineering Process**:

   Definition und Verbesserung von Software-Entwicklungsprozessen

9. **Software Engineering Tools and Methods**:

   Werkzeuge und Methoden für die Software-Entwicklung

10. **Software Quality**:

    Messen und Verbessern der Software-Qualität

Aufteilung auf Lehrveranstaltungen

- **Software-Engineering - Einführung**
  - ⇨ Software Requirements
  - ⇨ Software Design
  - ⇨ Software Construction
  - ⇨ Software Engineering Tools and Methods
- **Software-Engineering - Wartung und Qualitätssicherung**
  - ⇨ Software Testing
  - ⇨ Software Maintenance
  - ⇨ Software Configuration Management
  - ⇨ Software Engineering Management
  - ⇨ Software Engineering Process
  - ⇨ Software Engineering Tools and Methods
  - ⇨ Software Quality

Zielsetzungen der Vorlesung:

- ☺ Zuhörer für die Realität der Software-Entwicklung fit machen
- ☺ Techniken und Werkzeuge zum Umgang mit bestehender Software vermitteln
- ☺ Überblick über systematische Software-Analyse- und Testverfahren geben

Zielsetzungen der Übung:

- ☺ praktische Vertiefung der Lehrinhalte an einem realen Beispiel
- ☺ Konfrontation mit kommerziellen und „Open Source“-Werkzeugen

Inhaltsverzeichnis der Vorlesung - 1

1. Software-Entwicklung, -Wartung und (Re-)Engineering 15
   - 1.1 Einleitung
   - 1.2 Software-Qualität
   - 1.3 Iterative Softwareentwicklung
   - 1.4 Forward-, Reverse- und Reengineering
   - 1.5 Zusammenfassung

2. Konfigurationsmanagement 55
   - 2.1 Einleitung
   - 2.2 Versionsmanagement
   - 2.3 Variantenmanagement
   - 2.4 Releasemanagement
   - 2.5 Buildmanagement
   - 2.6 Änderungsmanagement
   - 2.7. Verteilte Softwareentwicklung **(plus Industrievortrag)**
   - 2.8 Zusammenfassung

Inhaltsverzeichnis der Vorlesung - 2

3. Statische Programmanalysen und Metriken..... 153

3.1 Einleitung

3.2 Softwarearchitekturen und -visualisierung

3.3 Kontroll- und datenflussorientierte Analysen

3.4 Softwaremetriken

3.5 Zusammenfassung

4. Dynamische Programmanalysen und Testen..... 244

4.1 Einleitung

4.2 Laufzeit- und Speicherplatzverbrauchsmessungen

4.3 Funktionsorientierte Testverfahren

4.4 Kontrollflussbasierte Testverfahren

4.5 Datenflussbasierte Testverfahren

4.6. Testen objektorientierter Programme **(plus Vortrag zu modellbasiertem Testen)**

4.7. Testmanagement und Testwerkzeuge **(plus zwei Industrievorträge)**

4.8 Zusammenfassung

Inhaltsverzeichnis der Vorlesung - 3

5. Management der Software-Entwicklung 357

5.1 „Neuere“ Vorgehensmodelle

5.2 Rational Unified Process für UML

5.3 Leichtgewichtige Prozessmodelle

5.4 Verbesserung der Prozessqualität

5.5 Projektpläne und Projektorganisation

5.6 Weitere Literatur

Übungen zur Vorlesung:

Ihre (fiktive) Aufgabe ist es, die Software eines „Open Source“-Projektes weiterzuentwickeln (ImageJ unter http://rsbweb.nih.gov/ij/).

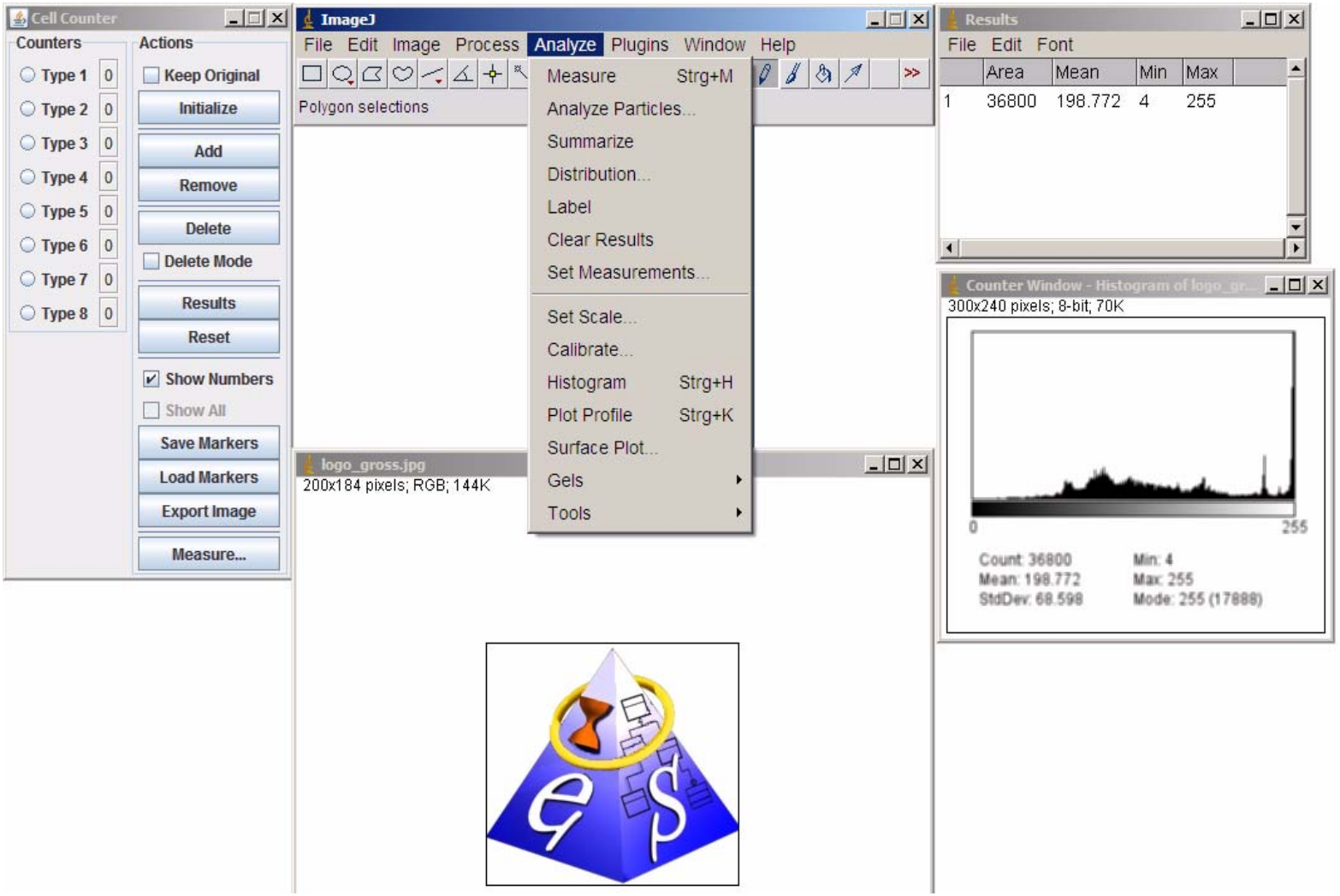
Vorgehensweise:

- Organisation als Teams von jeweils etwa 4 Studenten
- in Präsenzübungen werden Musterlösungen für Übungsaufgaben vorgestellt
- es gibt meist klausurähnliche Hausaufgaben
- Einsatz verschiedener CASE-Tools für („Open Source“-)Softwareentwicklung
- zusätzliche Betreuungstermine im PC-Pool bei Bedarf

Betreuer:

☞ **Johannes Bürdek** (Johannes.Buerdek@es.tu-darmstadt.de)

Übungs-Beispiel ImageJ (Screendump):



Zertifizierung zum Software-Tester (übernommen von Dr. Falk Fraikin):

- **ASQF** = **A**rbeitskreis **S**oftware-**Q**ualität **F**ranken bietet in Zusammenarbeit mit dem **iSQI** = international Software Quality Institute „standardisierte“ Ausbildung (Foundation/Advanced/Diploma Level) zum **Certified Tester** an (mit Unterstützung durch Gesellschaft für Informatik)
- in Deutschland waren 2003 fünf Kursanbieter **akkreditiert** und es gab etwa 550 **zertifizierte** Tester (auf Foundation Level); inzwischen gibt es alleine 17 sogenannte „Premiumanbieter“ solcher Kurse
- angeblich bestehen ca. 80% der Prüfungsteilnehmer die Prüfung zum „Certified Tester Foundation Level“
- wir werden auch dieses Jahr wieder Studierenden der TU Darmstadt die Möglichkeit bieten, am Ende des Semester gegen ermäßigte Prüfungsgebühr hier in Darmstadt die Prüfung zum „Certified Tester“ abzulegen
- empfohlene Lehrmaterialien und weitere Links sind
  - ⇨ Lehrbuch hierzu [SL12]
  - ⇨ weitere Informationen unter http://www.certified-tester.de/

Wichtige Literaturquellen:

[Ba00] H. Balzert: *Lehrbuch der Software-Technik (Band 1): Software-Entwicklung*, Spektrum Akademischer Verlag (2000), 2-te Auflage, 1136 Seiten

Sehr umfangreiches und gut lesbares Nachschlagewerk mit CD. Auf der CD findet man Werkzeuge, Videos, Übungsaufgaben mit Lösungen (aber nicht unsere), …

[Ba98] H. Balzert: *Lehrbuch der Software-Technik (Band 2): Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Spektrum Akademischer Verlag (1998), 769 Seiten

Hier findet man fast alles über das Gebiet Software-Technik, was nicht bereits in [Ba00] abgehandelt wurde. Wieder ist eine CD mit den entsprechenden Werkzeugen, ... beigelegt.

[Be00] R.V. Beizer: *Testing Object-Oriented Systems - Models, Patterns, Tools*, Addison-Wesley (2000), 1191 Seiten

Umfassende Quelle zum angesprochenen Thema. Gut geschrieben und umbedingt empfehlenswert, falls man die Thematik „Testen“ vertiefen will.

[CFP06] B. Collins-Sussman, B.W. Fitzpatrick, C.M. Pilato: *Versionskontrolle mit Subversion*, O’Reilly (2006)

Subversion (SVN) ist der moderne Nachfolger von CVS.

[FB01] K. Fogel, M. Bar: ***Open Source-Projekte mit CVS***, mitp-Verlag (2002), 2-te überarbeitete Auflage, 428 Seiten

Dieses Buch liefert gut lesbar das notwendige Know-how für die Verwendung und Administration von CVS sowie eine Einführung in die Prinzipien des Managements von „Open Source“-Projekten

[He03] H. Herold: ***make - das Profitool zur automatischen Generierung von Programmen***, Addison Wesley (2003), 230 Seiten

Akzeptable Einführung in make, das „Open Source“-Werkzeug zur Automatisierung von Übersetzungs- und Programmgenerierungsprozessen.

[Li02] P. Liggesmeyer: ***Software-Qualität: Testen, Analysieren und Verifizieren von Software***, Spektrum Akademischer Verlag (2002), 523 Seiten

Ein Standardwerk zum Thema Software-Qualitätssicherung. Kapitel 3 und 4 der Vorlesung stützen sich vor allem darauf ab.

[PBL05] K. Pohl, G. Böckle, F. van der Linden: *Software Product Line Engineering - Foundations, Principles, and Techniques*, Springer Verlag 2005, 467 Seiten

Ein (Lehr-)Buch zum „Software-Produkt-Familien“ (Management von Software-Varianten).

[So01] I. Sommerville: *Software Engineering*, Addison-Wesley - Pearson Studium, 6. Auflage (2001), 711 Seiten

Ins Deutsch übersetztes Lehrbuch, das sehr umfassend alle wichtigen Themen der Software-Technik knapp behandelt. Empfehlenswert!

[SL12] A. Spillner, T. Linz: ***Basiswissen Softwaretest***, dpunkt.verlag (2012; 5. Auflage), 290 Seiten

Diesem Buch liegt der Lehrplan „Grundlagen des Testens“ für den ASQF Certified Tester, Foundation Level zugrunde.

[Wh00] B.A. White: *Software Configuration Management Strategies and Rational ClearCase*, Addison Wesley (2000), 305 Seiten

Clearcase war lange Zeit „das“ kommerzielle Pendant zu CVS; es besitzt wesentlich mehr Features, die Administration erfordert allerdings auch mehr Aufwand.

1. Software-Entwicklung, -Wartung und (Re-)Engineering

„Software-Systeme zu erstellen, die nicht geändert werden müssen, ist unmöglich. Wurde Software erst einmal in Betrieb genommen, entstehen neue Anforderungen und vorhandene Anforderungen ändern sich ... "

[So01]

Lernziele:

- **Wissen** über Entwicklungsprozesse und Qualitätssicherung **auffrischen**
- Probleme mit der **Pflege „langlebiger" Software** kennenlernen
- Begriffe „**Software-Wartung und -Evolution**" definieren können
- **Forward-, Reverse- und Reengineering** unterscheiden können
- Motivation für den Inhalt dieser Lehrveranstaltung

1.1 Einleitung

Zur Erinnerung - die **Geschichte der Software-Technik**:

- Auslöser für Einführung der Software-Technik war die **Software-Krise** von 1968
- Der Begriff "**Software Engineering**" wurde von F.L. Bauer im Rahmen einer Study Group on Computer Science der NATO geprägt
- Bahnbrechend war die NATO-Konferenz

  "**Working Conference on Software Engineering**"

  vom 7. - 10. Oktober 1968 in Garmisch
- Das Gebiet Software-Technik wird der praktischen Informatik zugeordnet, hat aber auch Wurzeln in der theoretischen Informatik
- Informatikübergreifende Aspekte spielen eine wichtige Rolle (wie Projektplanung, Organisation, Psychologie, …)

Definition des Begriffs „Software-Technik”:

Software Engineering = **Software-Technik** ist nach [Ka98]:

- ⇨ die Entwicklung
- ⇨ die Pflege und
- ⇨ der Einsatz

qualitativ hochwertiger Software unter Einsatz von

- ⇨ wissenschaftlichen Methoden
- ⇨ wirtschaftlichen Prinzipien
- ⇨ geplanten Vorgehensmodellen
- ⇨ Werkzeugen
- ⇨ quantifizierbaren Zielen

Bislang in „Software Engineering - Einführung“ kennengelernt:

Entwicklung von Software, aber nicht Pflege (und Einsatz).

40 Jahre nach Beginn der Software-Krise:

Standish Group (http://www.standishgroup.com) veröffentlich in regelmäßigen Abständen den sogenannten „Chaos Report“:

- 25% aller betrachteten IT-Projekte sind gescheitert
- 50% aller betrachteten IT-Projekte sind dabei zu scheitern
- 25% aller betrachteten IT-Projekte sind erfolgreich
  (wurden im geplanten Finanz- und Zeitrahmen beendet)

Hauptgründe für Scheitern von Projekten:

Nach wie vor unklare Anforderungen und Abhängigkeiten sowie Probleme beim **Änderungsmanagement**!!!

1.2 Software-Qualität

Ziel der Software-Technik ist die effiziente Entwicklung **messbar** qualitativ hochwertiger Software, die

- ⇨ gewünschte Funktionalität anbietet
- ⇨ benutzerfreundliche Oberfläche besitzt
- ⇨ korrekt bzw. zuverlässig arbeitet
- ⇨ effizient (ökonomisch) mit Hardwareressourcen umgeht
- ⇨ wartbar (und leicht änderbar) ist

FURPS-Modell von HP zur Software-Qualitätssteigerung:

- ⇨ **F**unctionality (richtige Funktionalität)
- ⇨ **U**sability (gute Benutzeroberfläche)
- ⇨ **R**eliability (zuverlässige Funktionsweise)
- ⇨ **P**erformance (schnelle Reaktionszeiten)
- ⇨ **S**upportability (gute Programmstruktur)

Prinzipien der Qualititätssicherung:

- **Qualitätszielbestimmung**: Auftraggeber und Auftragnehmer legen vor Beginn der Software-Entwicklung gemeinsames Qualitätsziel für Software-System mit nachprüfbarem Kriterienkatalog fest (als Bestandteil des abgeschlossenen Vertrags zur Software-Entwicklung)
- **quantitative Qualitätssicherung**: Einsatz automatisch ermittelbarer Metriken zur Qualitätsbestimmung (objektivierbare, ingenieurmäßige Vorgehensweise)
- **konstruktive Qualitätssicherung**: Verwendung geeigneter Methoden, Sprachen und Werkzeuge (Vermeidung von Qualitätsprobleme)
- **integrierte, frühzeitige analytische Qualitätssicherung**: systematische Prüfung aller erzeugten Dokumente (Aufdeckung von Qualitätsproblemen)
- **unabhängige Qualitätssicherung**: Entwicklungsprodukte werden durch eigen-ständige Qualitätssicherungsabteilung überprüft und abgenommen (verhindert u.a. Verzicht auf Testen zugunsten Einhaltung des Entwicklungszeitplans)

Konstruktives Qualitätsmanagement zur Fehlervermeidung:

- **technische Maßnahmen**:
  - ⇨ Sprachen (wie z.B. UML für Modellierung, Java für Programmierung)
  - ⇨ Werkzeuge (UML-CASE-Tool wie Together oder …)
- **organisatorische Maßnahmen**:
  - ⇨ Richtlinien (Gliederungsschema für Pflichtenheft, Programmierrichtlinien)
  - ⇨ Standards (für verwendete Sprachen, Dokumentformate, Management)
  - ⇨ Checklisten (wie z.B. „bei Ende einer Phase müssen folgende Dokumente vorliegen” oder “Softwareprodukt erfüllt alle Punkte des Lastenheftes”)

Einhaltung von Richtlinien, Standards und Überprüfung von Checklisten kann durch Werkzeugeinsatz = technische Maßnahmen erleichtert (erzwungen) werden.

Beispiel für die Wichtigkeit konstruktiver Qualitätssicherung:

Das Mariner-Unglück: Mariner 1 (Venus-Sonde) musste 4 Minuten nach dem Start wegen unberechenbarem Flugverhalten zerstört werden. Gerüchten zufolge war folgender Softwarefehler in einem Fortranprogramm schuld:

Korrektes Programm:

```
DO 3 i= 1,3
   ... (irgendwelche Befehle)
3 CONTINUE
```

Fehlerhaftes Programm:

```
DO 3 i=1.3
   ... (irgendwelche Befehle)
3 CONTINUE
```

Erläuterung:

- korrektes Programm ist eine Schleife, die dreimal ausgeführt wird (für i = 1 bis 3)
- falsches Programm ist eine Zuweisung: DO3i = 1.3
- in Fortran spielen nämlich Leerzeichen keine Rolle
- Variablen brauchen nicht deklariert werden (DO3i ist eine Real-Variable)
- es gibt keine reservierten Schlüsselwörter (wie etwa DO)

Schlussfolgerungen aus Mariner-Vorfall:

- durch systematisches Testen hätte man den Fehler vermutlich bei Testläufen (Simulationen) finden können
- eine „vernünftige“ Programmiersprache hätte die Verwendung eines Dezimal-punktes anstelle von Komma als Syntaxfehler zur Übersetzungszeit gefunden
- das Aufstellen von (durch Werkzeuge überprüfte) Programmierrichtlinien für Fortranprogramme hätte das Unglück auch vermieden:
  - ⇨ alle Variablen werden deklariert (obwohl es nicht notwendig ist)
  - ⇨ die Verwendung von Leerzeichen in Variablennamen wird verboten
  - ⇨ ebenso die Verwendung von Schlüsselwörtern in Variablennamen
- ...

Analytisches Qualitätsmanagement zur Fehleridentifikation:

- **analysierende Verfahren**:

  der „Prüfling” (Programm, Modell, Dokumentation) wird von Menschen oder Werkzeugen auf Vorhandensein/Abwesenheit von Eigenschaften untersucht

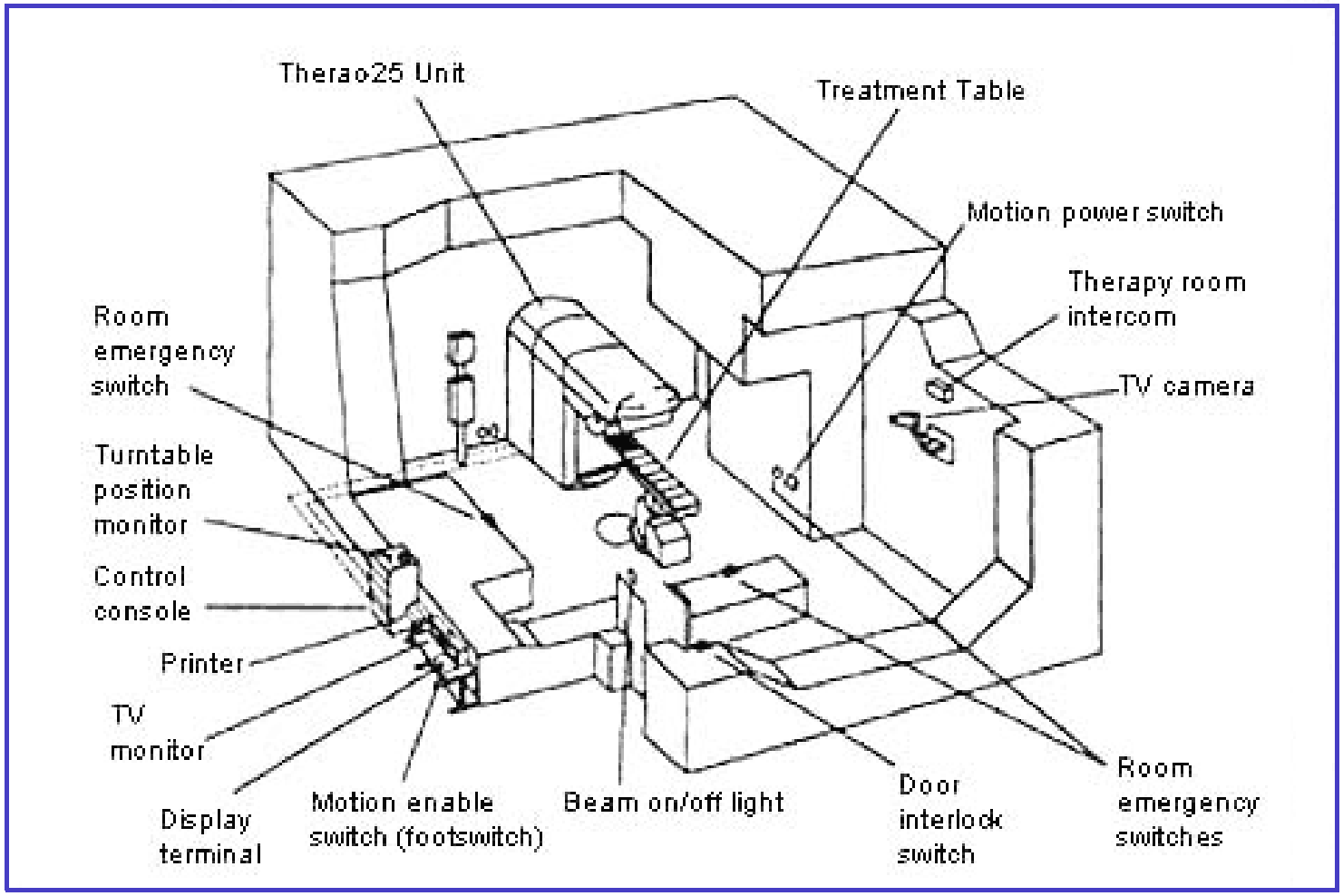
  - ⇨ **Inspektion** (Review, Walkthrough): organisiertes „Durchlesen” in Gruppe
  - ⇨ **statische Analyse**: werkzeuggestützte Ermittlung von „Anomalien”
  - ⇨ **Verifikation**: werkzeuggestützter Beweis von Eigenschaften

- **testende Verfahren**:

  der „Prüfling” wird mit konkreten oder abstrakten Eingabewerten von Menschen oder Werkzeugen ausgeführt

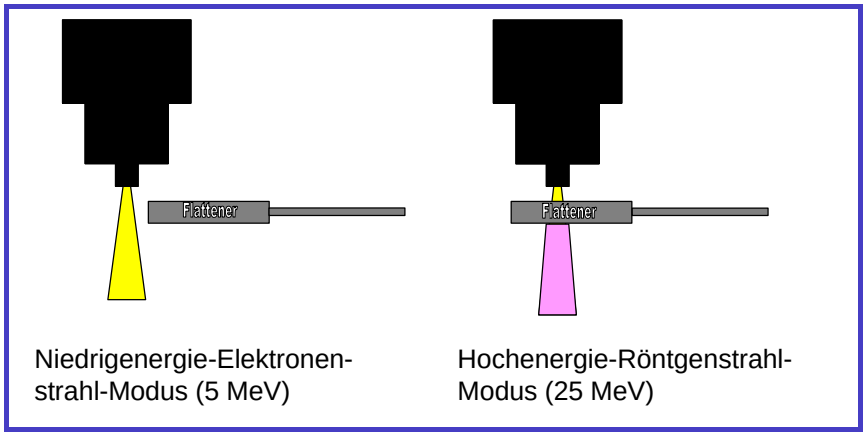
  - ⇨ **dynamischer Test** (Simulation): Ausführung mit konkreten Eingaben
  - ⇨ **symbolischer Test**: Interpretation mit symbolischen Eingaben (die oft unendliche Mengen möglicher konkreter Eingaben repräsentieren)

Beispiel für Wichtigkeit analytischer Qualitätssicherung - Therac 25:



Therac 25 - ein rein „softwaregesteuertes“ Bestrahlungsgerät:

Mindestens 6 Personen erhalten zwischen 1985 und 1987 größtenteils tödliche Überdosis bei Krebstherapie mit Strahlenbehandlung, da:



- ☹ Maschine hat zwei Betriebsmodi: Niedrigenergie- und Hochenergiebestrahlung
- ☹ tödliche Bestrahlung im Hochenergiemodus wird allein durch „Flattener“ verhindert, der Strahl modifiziert
- ☹ in mehreren Fällen war „Flattener“ bei Hochenergiebestrahlung falsch positioniert

Externe Sicht auf einen Unglücksfall:

- ☹ Techniker tippt „x“ für „X-Ray-Behandlung ein - anstatt „e“ für „electron mode“
- ☹ Techniker entdeckt Fehler allerdings sofort und korrigiert Fehleingabe sehr schnell und startet Behandlung
- ☹ auf dem Bildschirm wird die Fehlermeldung „Malfunction 54“ ausgegeben
- ☹ ähnliche Fehlermeldungen hatten bislang immer vorzeitigen Abbruch einer Behandlung ohne Bestrahlung angezeigt
- ☹ Techniker wiederholt also den Vorgang (Behandlung) zweimal
- ☹ Patient wird dreimal mit 25 MeV bestrahlt und stirbt 4 Monate später
- ☹ Techniker beharrt hartnäckig darauf, dass Therac-25 eine Fehlfunktion hatte, diese lässt sich aber zunächst nicht reproduzieren
- ☹ Herstellerfirma streitet das wohl zunächst längere Zeit ab; weitere Todesfälle sind die Folge

Interne Sicht auf den Vorfall:

- Intensität des Elektronenstrahls und Position des „Flatteners“ werden allein durch Software kontrolliert
- zu schnelle Eingabe der Daten führte dazu, dass
  - ⇨ „Flattener“ zwar aus Strahl entfernt wurde
  - ⇨ Intensität des Elektronenstrahls aber nicht korrigiert wurde
- erkannter Fehlerzustand führte zu kryptischer Fehlermeldung, aber nicht zur Blockade/Abbruch der Behandlung
- allein (fehlerhaft implementierte) Software-Locks sollen verhindern, dass Elektronenstrahl mit 25 MeV ohne „Flattener“ den Patienten trifft:
  - ⇨ Sperre wird durch Inkrementieren eines Bytes um 1 gesetzt
  - ⇨ Sperre wird durch Dekrementieren des Bytes um 1 rückgesetzt
  - ⇨ Sperrenüberprüfung erfolgt durch Test auf Wert (ungleich) „0“
  - ⇨ ein rekursiver Warteprozess führt zunächst beliebig viele Sperr-anforderungen aus, bevor er schließlich alle Sperren wieder freigibt

Fehleranalyse und -ursachen aus Managementsicht:

Übliche Sicherheitsanalyse für Therac-25 ging davon aus, dass die verwendete Software immer fehlerfrei funktionieren würde, weil (Zitate aus Bericht):

- „Programming errors have been reduced by extensive testing on a hardware simulator and under field conditions“
- „Program software does not degrade due to war, fatigue, or reproduction process“
- „Computer execution errors are caused by faulty hardware components and ... “

Die Wahrscheinlichkeit für „**Computer selects wrong energy**“ wurde deshalb auf 10^{-11} gesetzt (angenommener Zeitraum ist mir unbekannt).

Schwerpunkte „meiner“ Software Engineering-Vorlesungen:

Schwerpunkte von „Software Engineering - Einführung“:

- ⇨ Sprachen für die Entwicklung/Modellierung von Software
- ⇨ Werkzeuge zur Konstruktion von Software
- ⇨ Vorgehensmodelle für konstruktive Qualitätssicherung

Schwerpunkte von „Software Engineering - hier“:

- ⇨ Vorgehensmodelle für analytische Qualitätssicherung
- ⇨ Werkzeuge zur Überprüfung von Software(-Qualität)
- ⇨ Vorgehensweisen zur Verbesserung/Restrukturierung von Software
- ⇨ konstruktive Maßnahmen zum Management von Software-Entwicklungs- und Änderungsprozessen

1.3 Iterative Software-Entwicklung

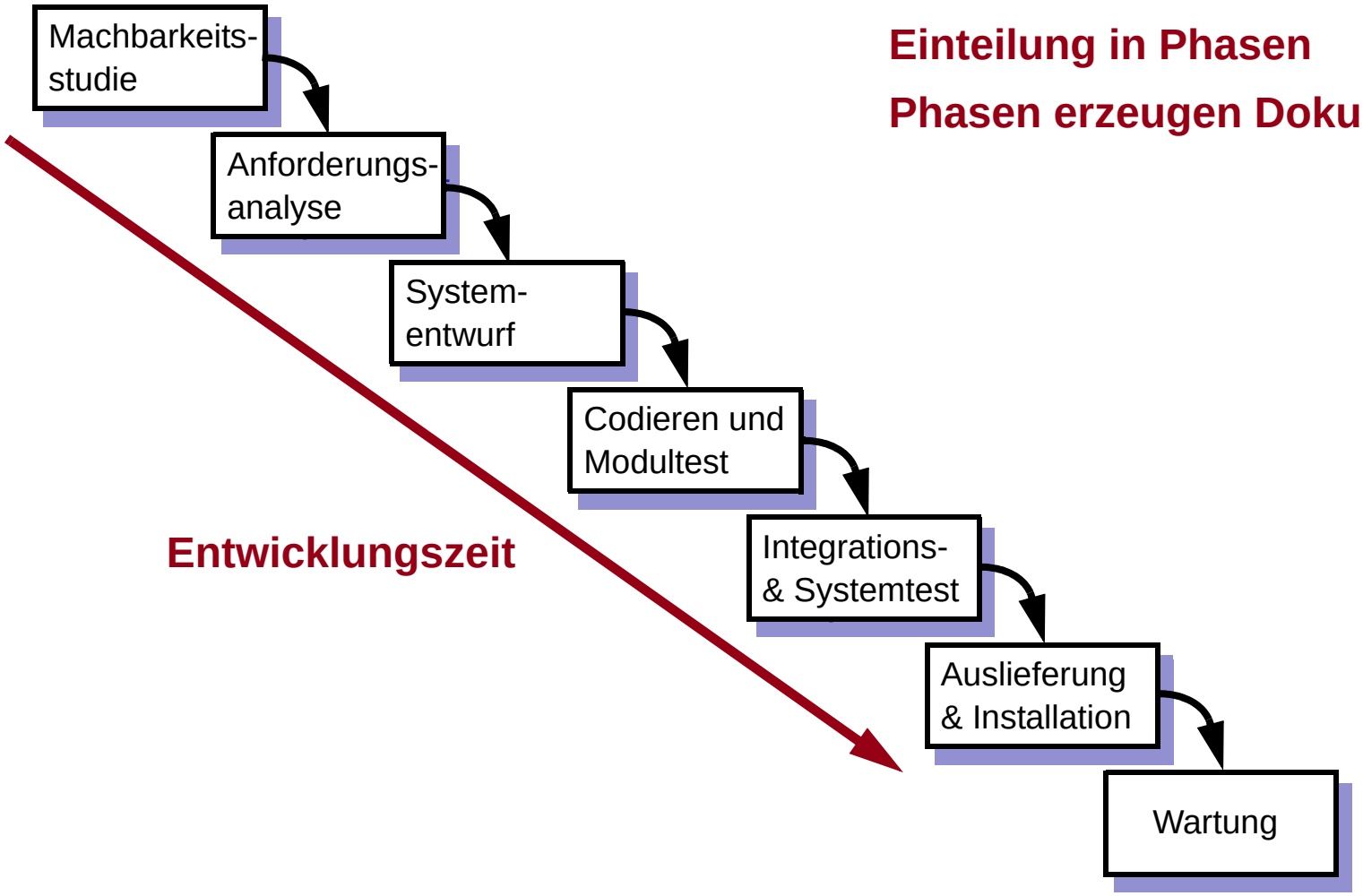
Voraussetzung für den sinnvollen Einsatz von Notationen und Werkzeugen zur Software-Entwicklung ist ein

- **Vorgehensmodell**, das den Gesamtprozeß der Software-Erstellung und -pflege in einzelne Schritte aufteilt.
- Zusätzlich müssen Verantwortlichkeiten der beteiligten Personen in Form von **Rollen** im Software-Entwicklungsprozess klar geregelt sein.

Zur Erinnerung:

- das „Wasserfallmodell“ war lange Zeit das Standardvorgehensmodell zur Erstentwicklung von Software
- in den letzten Jahren wurden für die (Weiter-)Entwicklung von Software bessere **iterative Vorgehensmodelle** entwickelt:
  - ⇨ V-Modell, Rational Unified Process, ...

Die Phasen des Wasserfallmodells im Überblick:



Einteilung in Phasen

Phasen erzeugen Dokumente

Machbarkeitsstudie (feasibility study):

Die Machbarkeitsstudie schätzt **Kosten und Ertrag der** geplanten **Software-Entwicklung** ab. Dazu grobe Analyse des Problems mit Lösungsvorschlägen.

- **Aufgaben**:
  - ⇨ Problem informell und abstrahiert beschreiben
  - ⇨ verschiedene Lösungsansätze erarbeiten
  - ⇨ Kostenschätzung durchführen
  - ⇨ Angebotserstellung
- **Ergebnisse**:
  - ⇨ Lastenheft = (sehr) grobes Pflichtenheft
  - ⇨ Projektkalkulation
  - ⇨ Projektplan
  - ⇨ Angebot an Auftraggeber

Anforderungsanalyse (requirements engineering):

In der Anforderungsanalyse wird exakt festgelegt, **was die Software leisten soll**, aber nicht wie diese Leistungsmerkmale erreicht werden.

- **Aufgaben**:
  - genaue Festlegung der Systemeigenschaften wie Funktionalität, Leistung, Benutzungsschnittstelle, Portierbarkeit, … im Pflichtenheft
  - Bestimmen von Testfällen
  - Festlegung erforderlicher Dokumentationsdokumente
- **Ergebnisse**:
  - Pflichtenheft = Anforderungsanalysedokument
  - Akzeptanztestplan
  - Benutzungshandbuch (1-te Version)

Systementwurf (system design/programming-in-the-large):

Im Systementwurf wird exakt festgelegt, wie die Funktionen der Software zu realisieren sind. Es wird der **Bauplan der Software**, die Software-Architektur, entwickelt.

- **Aufgaben**:
  - ⇨ Programmieren-im-Großen = Entwicklung eines Bauplans
  - ⇨ Grobentwurf, der System in Teilsysteme/Module zerlegt
  - ⇨ Auswahl bereits existierender Software-Bibliotheken, Rahmenwerke, …
  - ⇨ Feinentwurf, der Modulschnittstellen und Algorithmen vorgibt
- **Ergebnisse**:
  - ⇨ Entwurfsdokument mit Software-Bauplan
  - ⇨ detailierte(re) Testpläne

Codieren und Modultest (programming-in-the-small):

Die eigentliche **Implementierungs- und Testphase**, in der einzelne Module (in einer bestimmten Reihenfolge) realisiert und validiert werden.

- **Aufgaben**:
  - ⇨ Programmieren-im-Kleinen = Implementierung einzelner Module
  - ⇨ Einhaltung von Programmierrichtlinien
  - ⇨ Code-Inspektionen kritischer Modulteile (Walkthroughs)
  - ⇨ Test der erstellten Module
- **Ergebnisse**:
  - ⇨ Menge realisierter Module
  - ⇨ Implementierungsberichte (Abweichungen vom Entwurf, Zeitplan, …)
  - ⇨ technische Dokumentation einzelner Module
  - ⇨ Testprotokolle

Integration und Systemtest:

Die einzelnen Module werden schrittweise zum **Gesamtsystem zusammengebaut**. Diese Phase kann mit der vorigen Phase verschmolzen werden, falls der Test isolierter Module nicht praktikabel ist.

- **Aufgaben**:
  - ⇨ Systemintegration = Zusammenbau der Module
  - ⇨ Gesamtsystemtest in Entwicklungsorganisation durch Kunden (α-Test)
  - ⇨ Fertigstellung der Dokumentation
- **Ergebnisse**:
  - ⇨ fertiges System
  - ⇨ Benutzerhandbuch
  - ⇨ technische Dokumentation
  - ⇨ Testprotokolle

Auslieferung und Installation:

Die Auslieferung (Installation) und **Inbetriebnahme** der Software **beim Kunden** findet häufig in zwei Phasen statt.

- **Aufgaben**:
  - ⇨ Auslieferung an ausgewählte (Pilot-)Benutzer (β-Test)
  - ⇨ Auslieferung an alle Benutzer
  - ⇨ Schulung der Benutzer
- **Ergebnisse**:
  - ⇨ fertiges System
  - ⇨ Akzeptanztestdokument

Wartung (Maintenance):

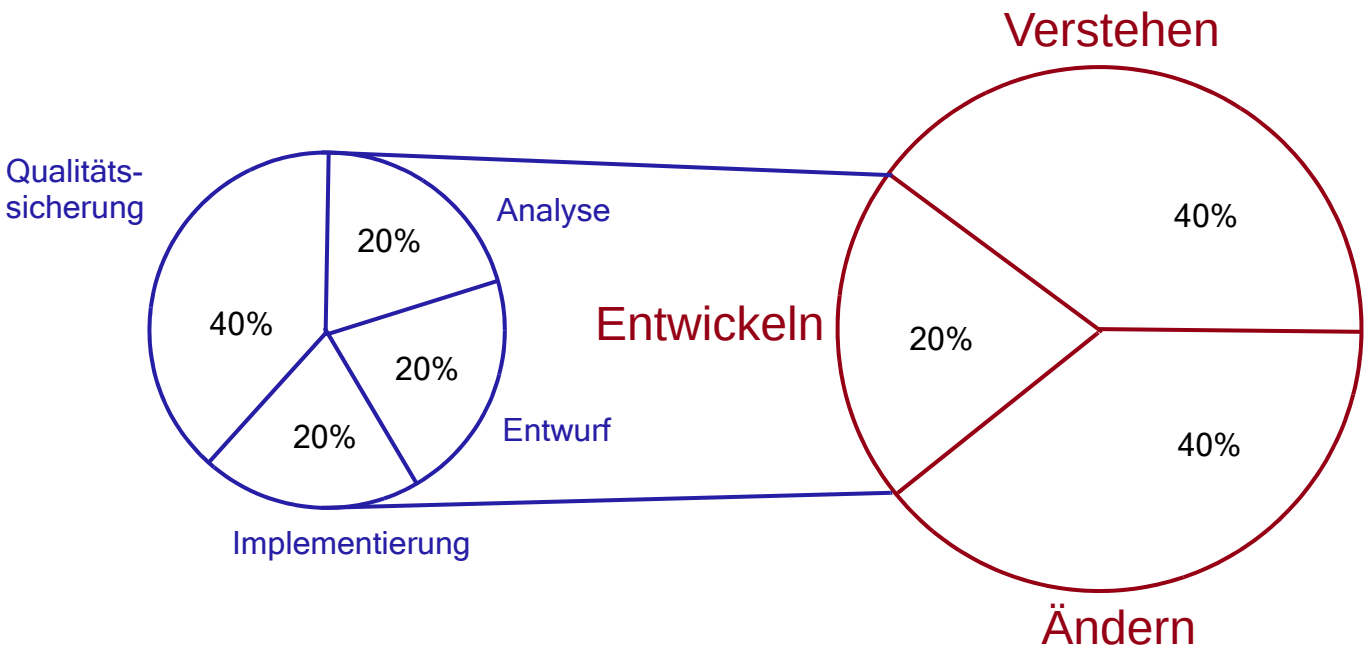
Nach der ersten Auslieferung der Software an die Kunden beginnt das Elend der Software-Wartung, das ca. 60% der gesamten Software-Kosten ausmacht.

- **Aufgaben**:
  - ca. 20% Fehler beheben (corrective maintenance)
  - ca. 20% Anpassungen durchführen (adaptive maintenance)
  - ca. 50% Verbesserungen vornehmen (perfective maintenance)
- **Ergebnisse**:
  - Software-Problemberichte (bug reports)
  - Software-Änderungsvorschläge
  - neue Software-Versionen

Probleme mit dem Wasserfallmodell:

- ☹ zu Projektbeginn sind nur ungenaue Kosten- und Ressourcenschätzungen möglich
- ☹ ein Pflichtenheft kann nie den Umgang mit dem fertigen System ersetzen, das erste sehr spät entsteht (Risikomaximierung)
- ☹ es gibt Fälle, in denen zu Projektbeginn kein vollständiges Pflichtenheft erstellt werden kann (weil Anforderungen nicht klar)
- ☹ Anforderungen werden früh eingefroren, notwendiger Wandel (aufgrund organisatorischer, politischer, technischer, … Änderungen) nicht eingeplant
- ☹ strikte Phaseneinteilung ist unrealistisch (Rückgriffe sind notwendig)
- ☹ Wartung mit ca. 60% des Gesamtaufwandes ist eine Phase

Andere Darstellung der Aufwandsverteilung:



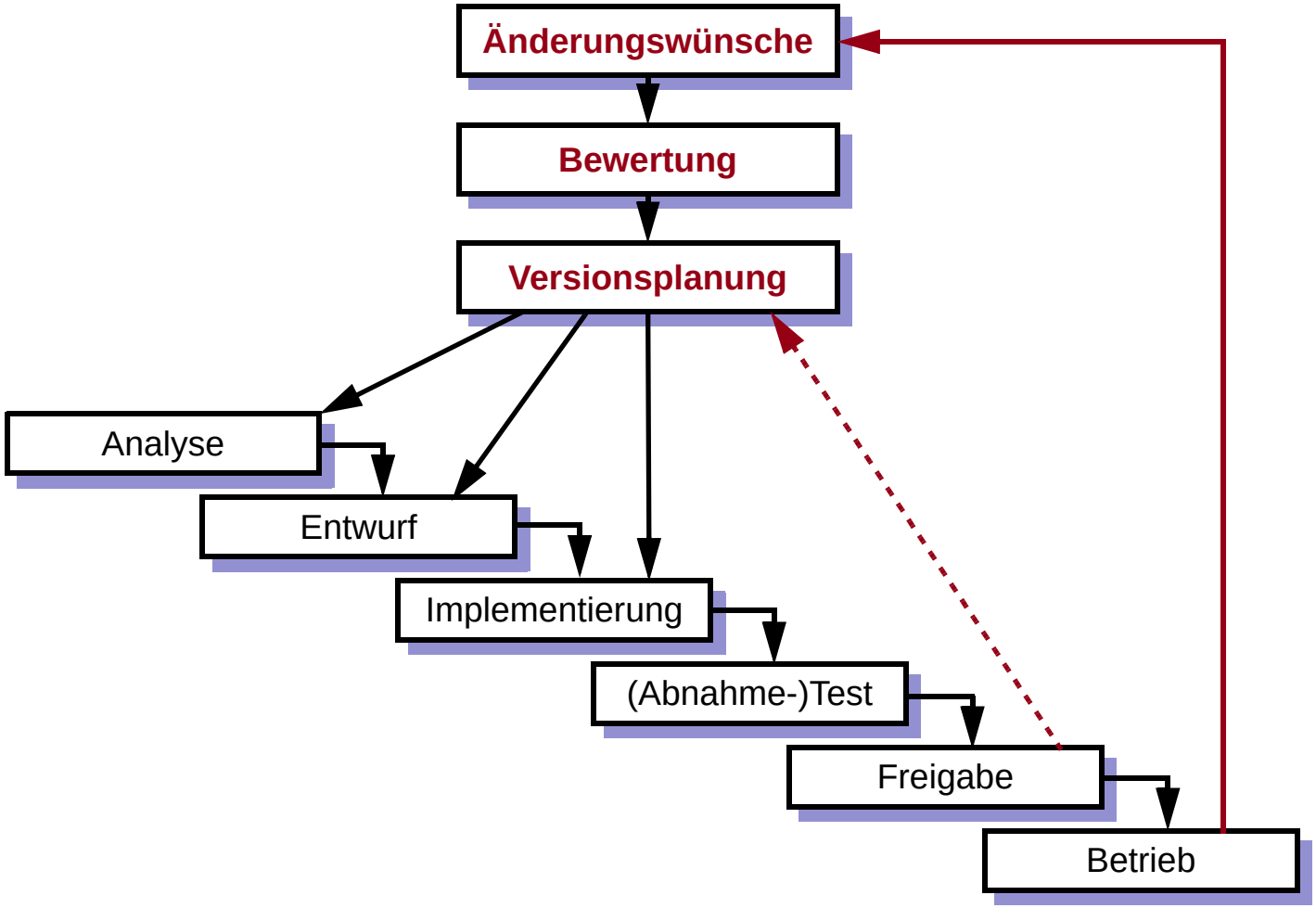
nach Nosek und Palvia, 1990

⇨ wenn man „Verstehen“ (bestehenden Codes) ganz dem Bereich Software-Wartung zuschlägt, kommt man sogar auf **80% Wartungsaufwand**

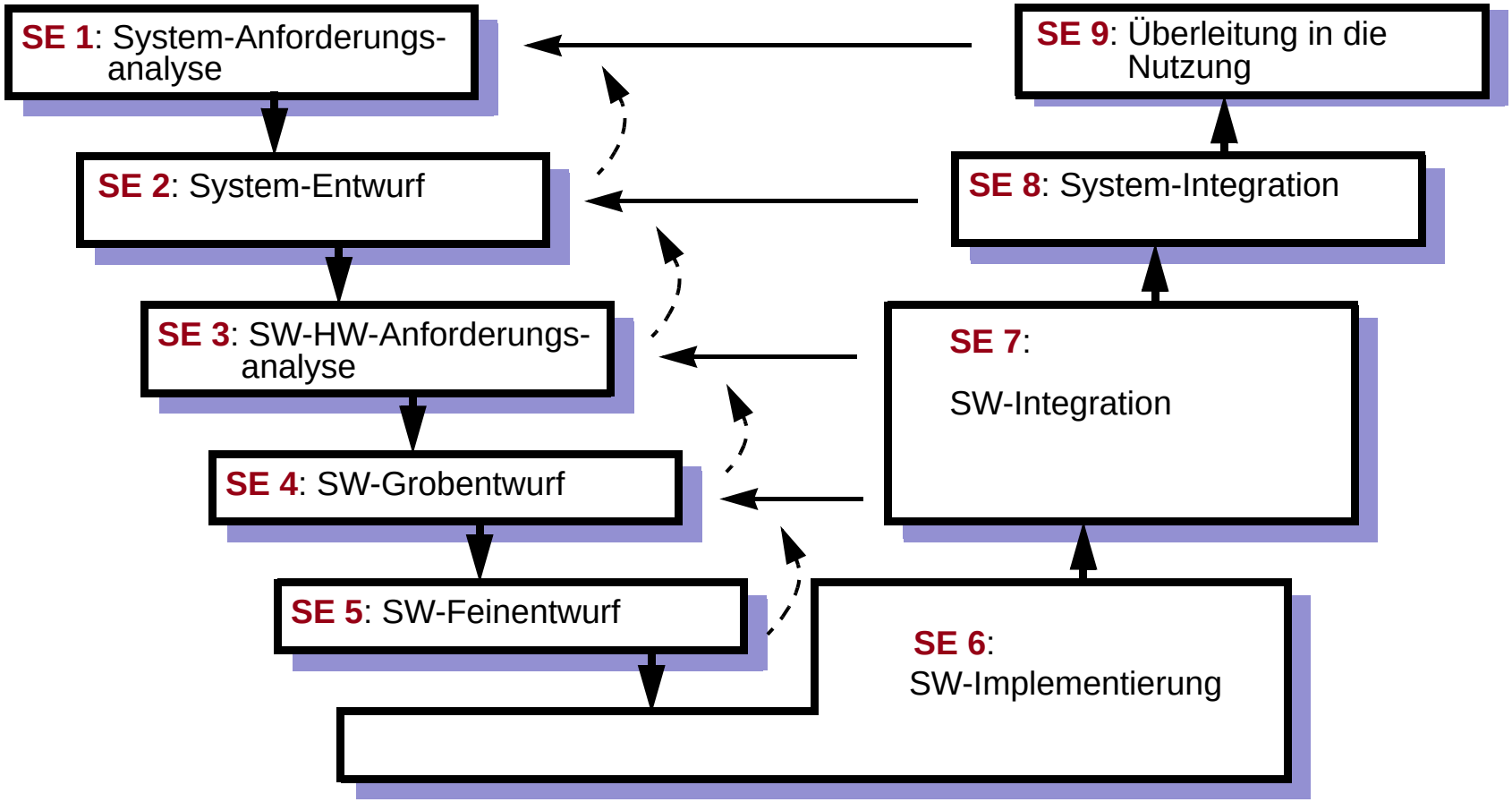
Typische Probleme in der Wartungsphase:

- Einsatz wenig erfahrenen Personals (nicht Entwicklungspersonal)
- Fehlerbehebung führt neue Fehler ein
- stetige Verschlechterung der Programmstruktur
- Zusammenhang zwischen Programm und Dokumentation geht verloren
- zur Entwicklung eingesetzte Werkzeuge (CASE-Tools, Compiler, …) sterben aus
- benötigte Hardware steht nicht mehr zur Verfügung
- Resourcenkonflikte zwischen Fehlerbehebung und Anpassung/Erweiterung
- völlig neue Ansprüche an Funktionalität und Benutzeroberfläche
- ...

Einfaches Software-Lebenszyklus-Prozessmodell für die Wartung:



Das V-Modell (Standard der Bundeswehr bzw. aller Bundesbehörden):



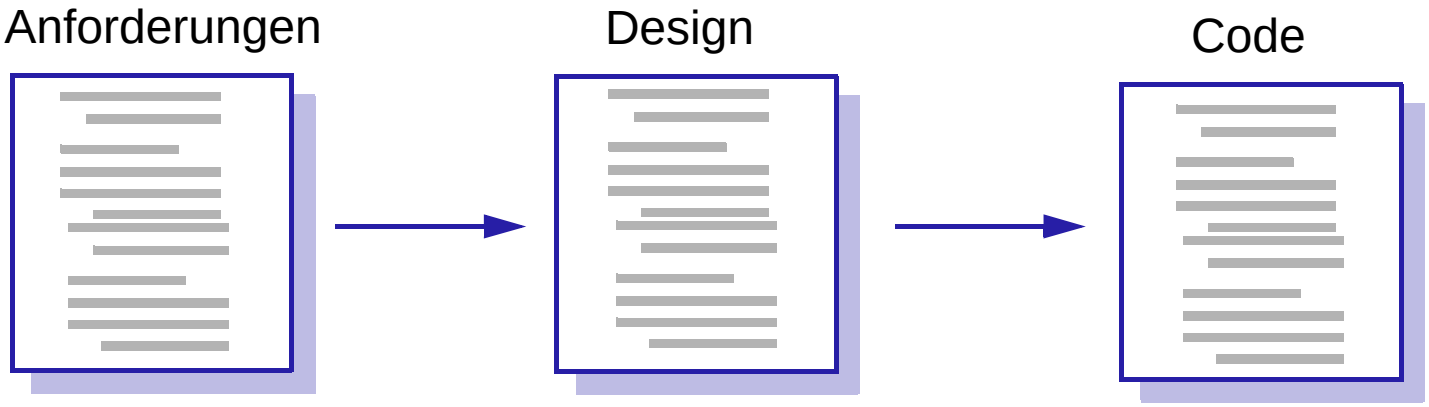
☞ weitere Informationen zu Prozessmodellen in „Kapitel 5 der Vorlesung“

Beschreibung der einzelnen Phasen des V-Modells:

- **Systemanforderungsanalyse**: Gesamtsystem einschließlich aller Nicht-DV-Komponenten wird beschrieben (fachliche Anforderungen und Risikoanalyse)
- **Systementwurf**: System wird in technische Komponenten (Subsysteme) zerlegt, also die Grobarchitektur des Systems definiert
- **Softwareanforderungsanalyse**: technischen Anforderungen an die bereits identifizierten Komponenten werden definiert
- **Softwaregrobentwurf**: Softwarearchitektur wird bis auf Modulebene festgelegt
- **Softwarefeinentwurf**: Details einzelner Module werden festgelegt
- **Softwareimplementierung**: wie beim Wasserfallmodell (inklusive Modultest)
- **Software-/Systemintegration**: schrittweise Integration und Test der verschiedenen Systemanteile
- **Überleitung in die Nutzung**: entspricht Auslieferung bei Wasserfallmodell

1.4 Forward-, Reverse- und Reengineering

Beim **Forward Engineering** ist das fertige Softwaresystem das Ergebnis des Entwicklungsprozesses. Ausgehend von Anforderungsanalyse (Machbarkeitsstudie) wird ein neues Softwaresystem entwickelt:



- ⇨ Anforderungs- und Design-Dokumente für Code existieren (hoffentlich)
- ⇨ alle Dokumente sind - da voneinander abgeleitet - (noch) konsistent
- ⇨ auf Entwickler des Codes kann (noch) zugegriffen werden

Wunsch und Wirklichkeit der Software-Evolution:

„Software-Systeme zu erstellen, die nicht geändert werden müssen, ist unmöglich. Wurde Software erst einmal in Betrieb genommen, entstehen neue Anforderungen und vorhandene Anforderungen ändern sich ... "

[So01]

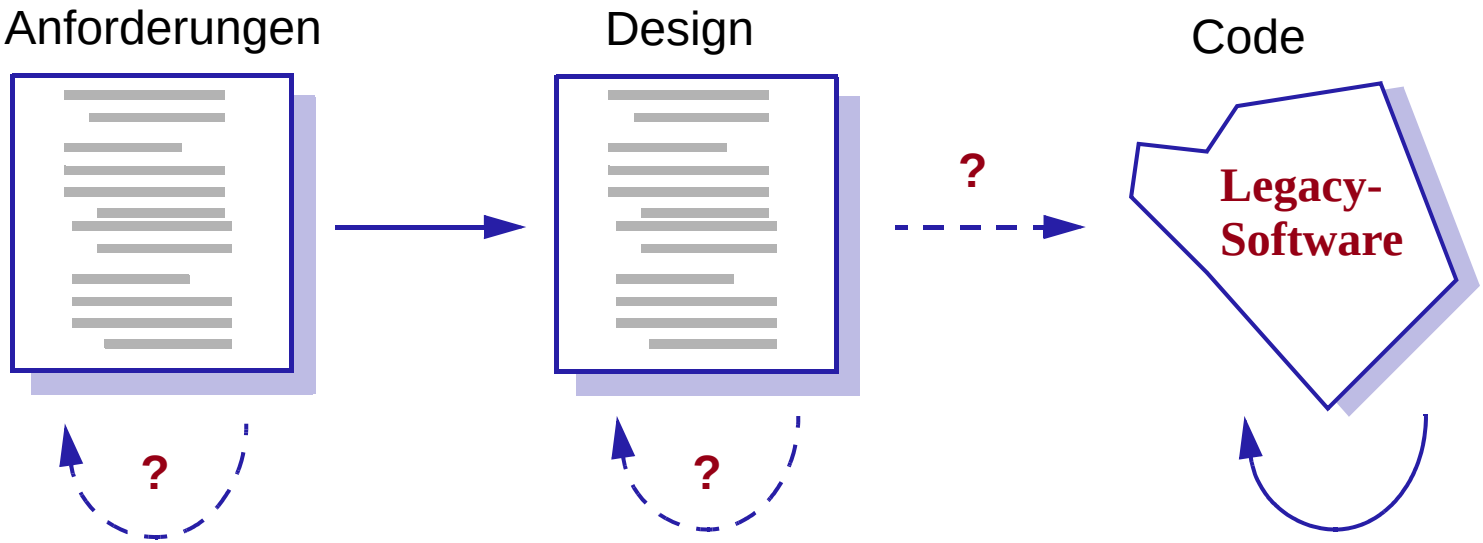
Software-Evolution - Wünsche:

- ☺ Wartung ändert Software kontrolliert ohne Design zu zerstören
- ☺ Konsistenz aller Dokumente bleibt erhalten

Software-Evolution - Wirklichkeit:

- ☹ ursprüngliche Systemstruktur wird ignoriert
- ☹ Dokumentation wird unvollständig oder unbrauchbar
- ☹ Mitarbeiter verlassen Projekt

Ergebnis unkontrollierter Software-Evolution:

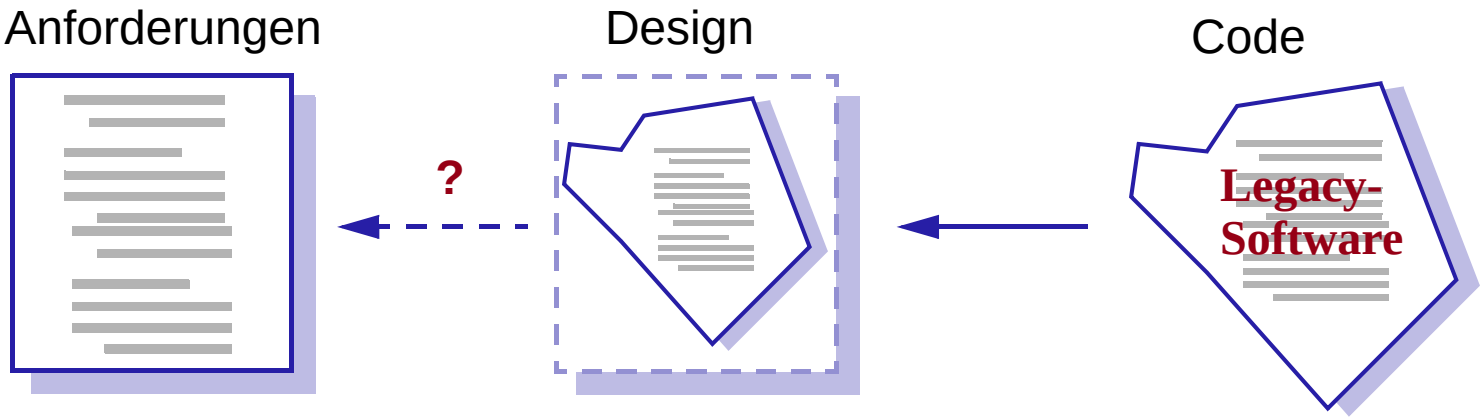


Legacy = „wertvolles“ Erbe

- ☹ der Code ändert sich mit Sicherheit, seine Struktur bleibt meist erhalten (auch wenn sie neuen Anforderungen eigentlich nicht gewachsen ist)
- ☹ das Design wird nach einer gewissen Zeit nicht mehr aktualisiert
- ☹ die Anforderungsdokumente werden erst recht nicht mehr gepflegt

Reverse Engineering von „Legacy Software“:

Beim **Reverse Engineering** ist das vorhandene Software-System der Ausgangspunkt der Analyse. Ausgehend von existierender Implementierung wird meist „nur“ das Design rekonstruiert und dokumentiert. Es wird (noch) nicht das betrachtete System modifiziert.



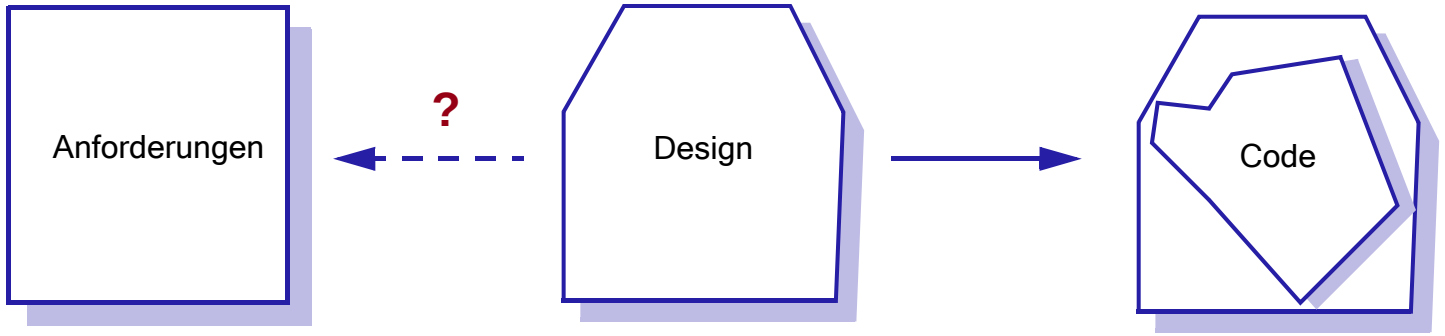
Fragen:

- Lässt sich das Software-System noch restrukturieren/sanieren?
- Oder kann man sein Innenleben „verkapseln“ und zunächst weiterverwenden?

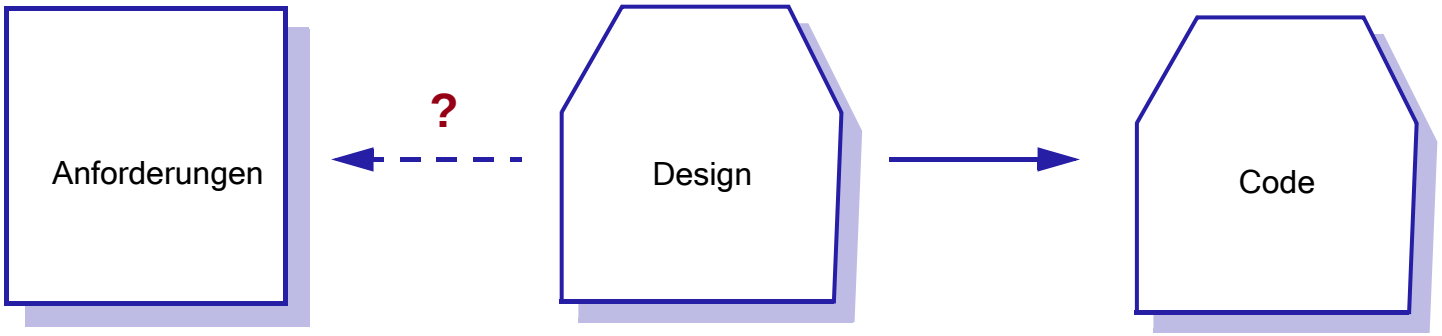
Strategien des Reengineerings:

Reengineering befaßt sich mit der Sanierung eines vorhandenen Software-Systems bzw. seiner Neuimplementierung. Dabei werden die Ergebnisse des Reverse Engineerings als Ausgangspunkt genommen

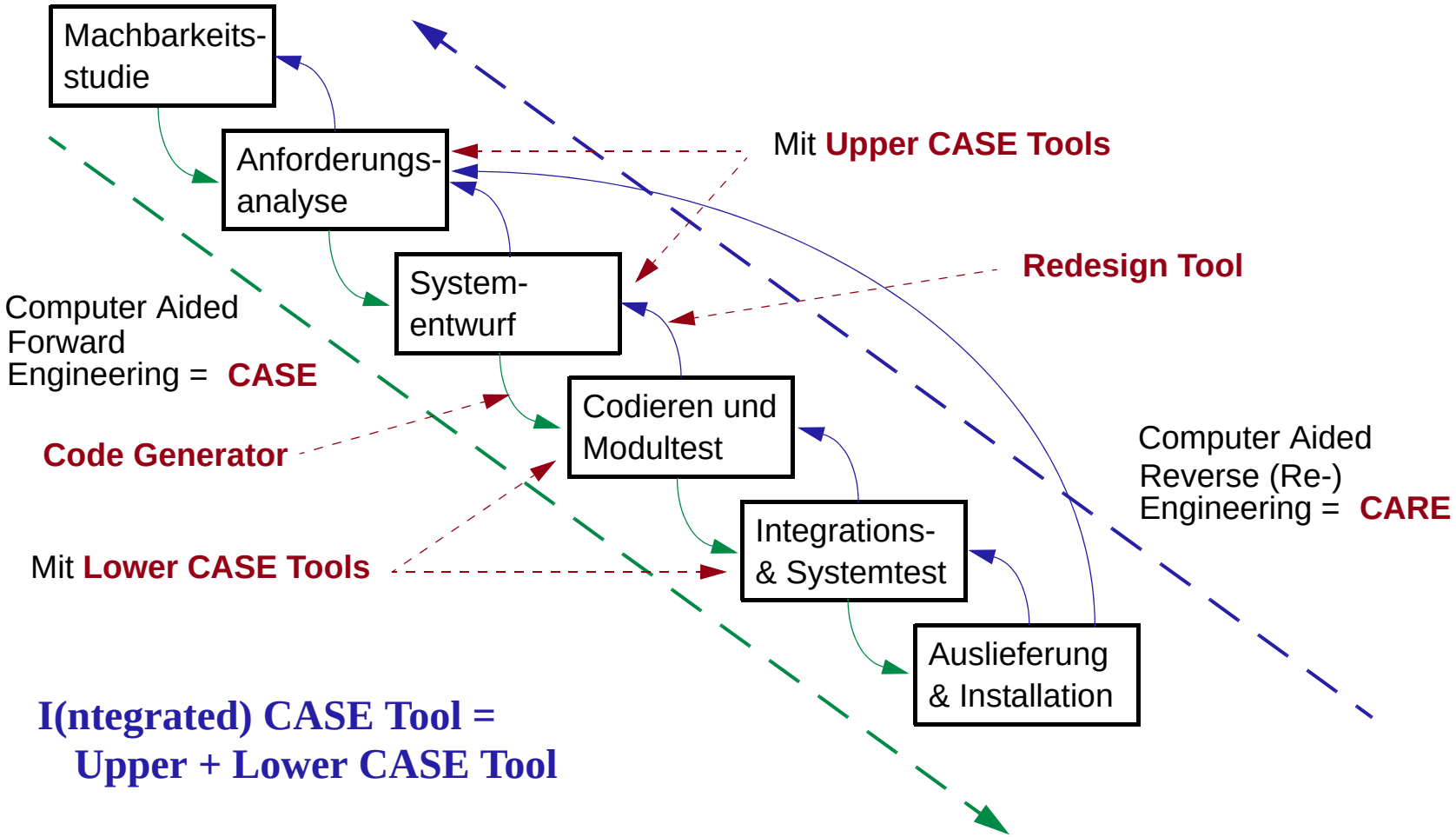
Restrukturierung des Designs, Kapselung des Codes:



Restrukturierung von Design und Code:



Forward- und Re(-verse)-engineering = Round Trip Engineering:



I(ntegrated) CASE Tool =
Upper + Lower CASE Tool

1.5 Zusammenfassung

Die Lehrveranstaltung „Software Engineering - Einführung“ und das „SW-Praktikum“ haben sich nur mit dem **Forward Engineering** von Software-Systemen befasst, also nur mit ca. 20% - 40% des Software-Lebenszyklus. Das Thema „**Software-Qualitäts-sicherung**“ wurde zudem nur kurz angerissen.

In dieser Vorlesung befassen wir uns deshalb mit:

- Kapitel 2: Management von Software-Änderungsprozessen
- Kapitel 3: Analyse und Überwachung von Software(-Qualität)
- Kapitel 4: Qualitätssicherung durch systematisches Testen
- Kapitel 5: Management der Software-Entwicklung

1.6 Zusätzliche Literatur

[BD00] B. Bruegge, A.H. Dutoit: *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall (2000)

[BEM92] A.W. Brown, A.N. Earl, J.A. McDermid: *Software Engineering Environments: Automated Support for Software Engineering*, McGraw-Hill (1992)

[Di72] E.W. Dijkstra: *The Humble Programmer*, Communications of the ACM, Vol. 15, No. 10 (1972)

[Fu93] A. Fugetta: *A Classification of CASE Technology*, Computer, Vol. 26, No. 12, S. 25-38, IEEE Computer Society Press (1993)

[GJ96] P.K. Garg, M. Jazayeri (Eds.): *Process-Centered Software Engineering Environments*, IEEE Computer Society Press (1996)

[Hr00] P. Hruschka: *Mein Weg zu CASE: von der Idee über Methoden zu Werkzeugen*, Hanser Verlag (1991)

[IEEE83] IEEE: *Standard Glossar of Software Engineering Terminology - IEEE Standard 729*, IEEE Computer Society Press (1983)

[Jo92] C. Jones: *CASE's Missing Elements*, IEEE Spektrum, Juni 1992, S. 38-41, IEEE Computer Society Press (1992)

[Ka98] B. Kahlbrandt: *Software-Engineering: Objektorientierte Software-Entwicklung mit der Unified Modeling Language*, Springer Verlag (1998)

[Na96] M. Nagl (ed.): *Building Thightly Integrated Software Development Environments: The IPSEN Approach*, LNCS 1170, Springer Verlag (1996)

[Ro92] Ch. Roth: *Die Auswirkungen von CASE*, Proc. GI-Jahrestagung 1992, Karlsruhe, Informatik aktuell, S. 648-656

[Sn87] H.M. Sneed: *Software-Management*, Müller GmbH (1987)

2. Konfigurationsmanagement

„Beim Konfigurationsmanagement handelt es sich um die Entwicklung und Anwendung von Standards und Verfahren zur Verwaltung eines sich weiterentwickelnden Systemprodukts."

[So01]

Lernziele:

- verstehen, warum **Konfigurationsmanagement** (**KM**) wichtig ist
- Einführung in die wichtigsten **Aufgabengebiete** des KMs
- „Open Source" und kommerzielle **CASE-Werkzeuge** für KM kennenlernen
- KM für **verteilte Software-Entwicklung** in „Open Source"-Projekten erleben
- selbständig **KM-Planung** für (kleine) Softwareprodukte durchführen können

2.1 Einleitung

Behandelte Fragestellungen:

- ☹ Das System lief gestern noch; was hat sich seitdem geändert?
- ☹ Wer hat diese (fehlerhafte?) Änderung wann und warum durchgeführt?
- ☹ Wer ist von meinen Änderungen an dieser Datei betroffen?
- ☹ Auf welche Version des Systems bezieht sich die Fehlermeldung?
- ☹ Wie erzeuge ich Version x.y aus dem Jahre 1999 wieder?
- ☹ Welche Fehlermeldungen sind in dieser Version bereits bearbeitet?
- ☹ Welche Erweiterungswünsche liegen für das nächste Release vor?
- ☹ Die Platte ist hinüber; was für einen Status haben die Backups?
- ☹ …

Definition von Software-KM nach IEEE-Standard 828-1988 [IEEE98]:

SCM (Software Configuration Management) constitutes **good engineering practice** for all software projects, whether phased development, rapid prototyping, or ongoing maintenance. It enhances the reliability and quality of software by:

- Providing structure for **identifying and controlling** documentation, code, interfaces, and databases to support all life cycle phases
- Supporting a chosen **development/maintenance methodology** that fits the requirements, standards, policies, organization, and management philosophy
- Producing **management and product information** concerning the status of baselines, change control, tests, releases, audits etc.

Anmerkung:

- wenig konkrete Definition
- unabhängig von „Software“

Definition nach DIN EN ISO 10007:

„KM (Konfigurationsmanagement) ist eine Managementdisziplin, die über die gesamte Entwicklungszeit eines Erzeugnisses angewandt wird, um Transparenz und Überwachung seiner funktionellen und physischen Merkmale sicherzustellen. ...
Der KM-Prozess umfasst die folgenden integrierten Tätigkeiten:

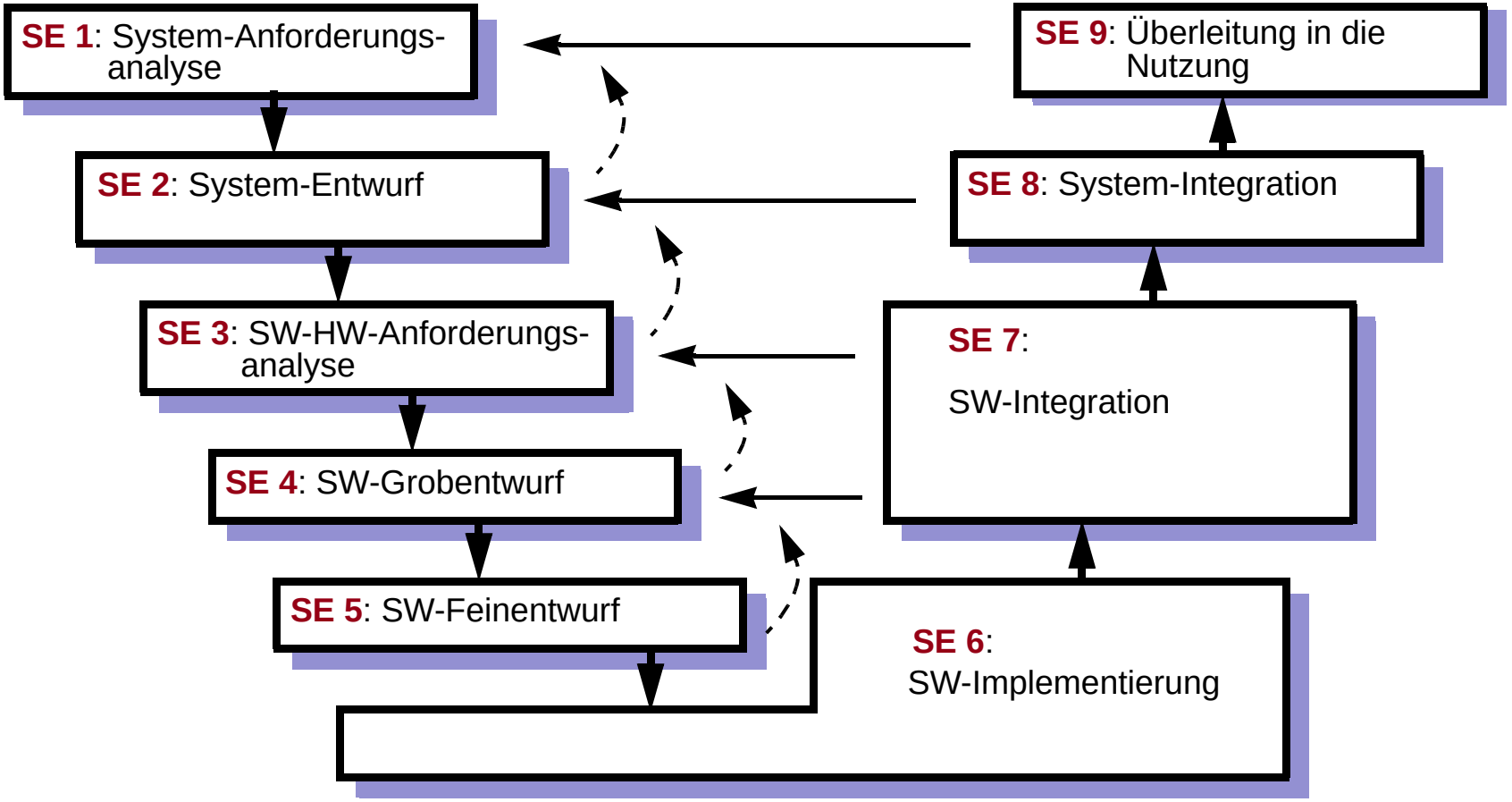
- **Konfigurationsidentifizierung**: Definition und Dokumentation der Bestandteile eines Erzeugnisses, Einrichten von Bezugskonfigurationen, ...
- **Konfigurationsüberwachung**: Dokumentation und Begründung von Änderungen, Genehmigung oder Ablehnung von Änderungen, Planung von Freigaben, ...
- **Konfigurationsbuchführung**: Rückverfolgung aller Änderungen bis zur letzten Bezugskonfiguration, ...
- **Konfigurationsauditierung**: Qualititätssicherungsmaßnahmen für Freigabe einer Konfiguration eines Erzeugnisses (siehe Kapitel 3 und Kapitel 4)
- **KM-Planung**: Festlegung der Grundsätze und Verfahren zum KM in Form eines KM-Plans

Werkzeugorientierte Sicht auf KM-Aktivitäten:

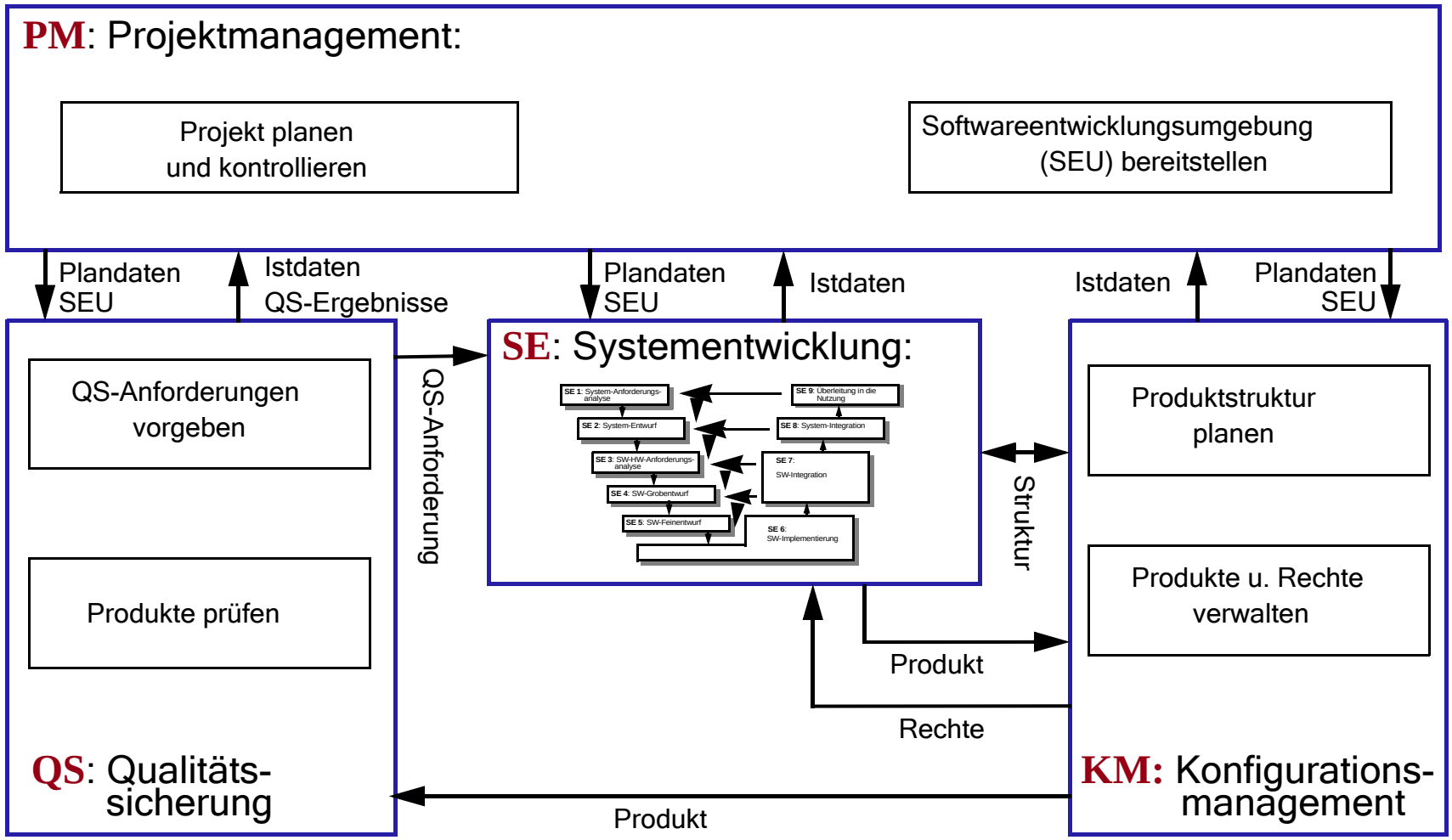
1. **KM-Planung**: Beschreibung der Standards, Verfahren und Werkzeuge, die für KM benutzt werden; wer darf/muss wann was machen (Abschnitt 2.1)
2. **Versionsmanagement**: Verwaltung der Entwicklungsgeschichte eines Produkts; also wer hat wann, wo, was und warum geändert (Abschnitt 2.2)
3. **Variantenmanagement**: Verwaltung parallel existierender Ausprägungen eines Produkts für verschiedene Anforderungen, Länder, Plattformen (Abschnitt 2.3)
4. **Releasemanagement**: Verwaltung und Planung von Auslieferungsständen; wann wird eine neue Produktversion mit welchen Features auf den Markt geworfen (Abschnitt 2.4)
5. **Buildmanagement**: Erzeugung des auszulieferenden Produkts; wann muss welche Datei mit welchem Werkzeug generiert, übersetzt, ... werden (Abschnitt 2.5)
6. **Änderungsmanagement**: Verwaltung von Änderungsanforderungen; also Bearbeitung von Fehlermeldungen und Änderungswünschen (Feature Requests) sowie Zuordnung zu Auslieferungsständen (Abschnitt 2.6)

© Prof. Dr. Andy Schürr (TU Darmstadt, FB 18, Institut für Datentechnik)

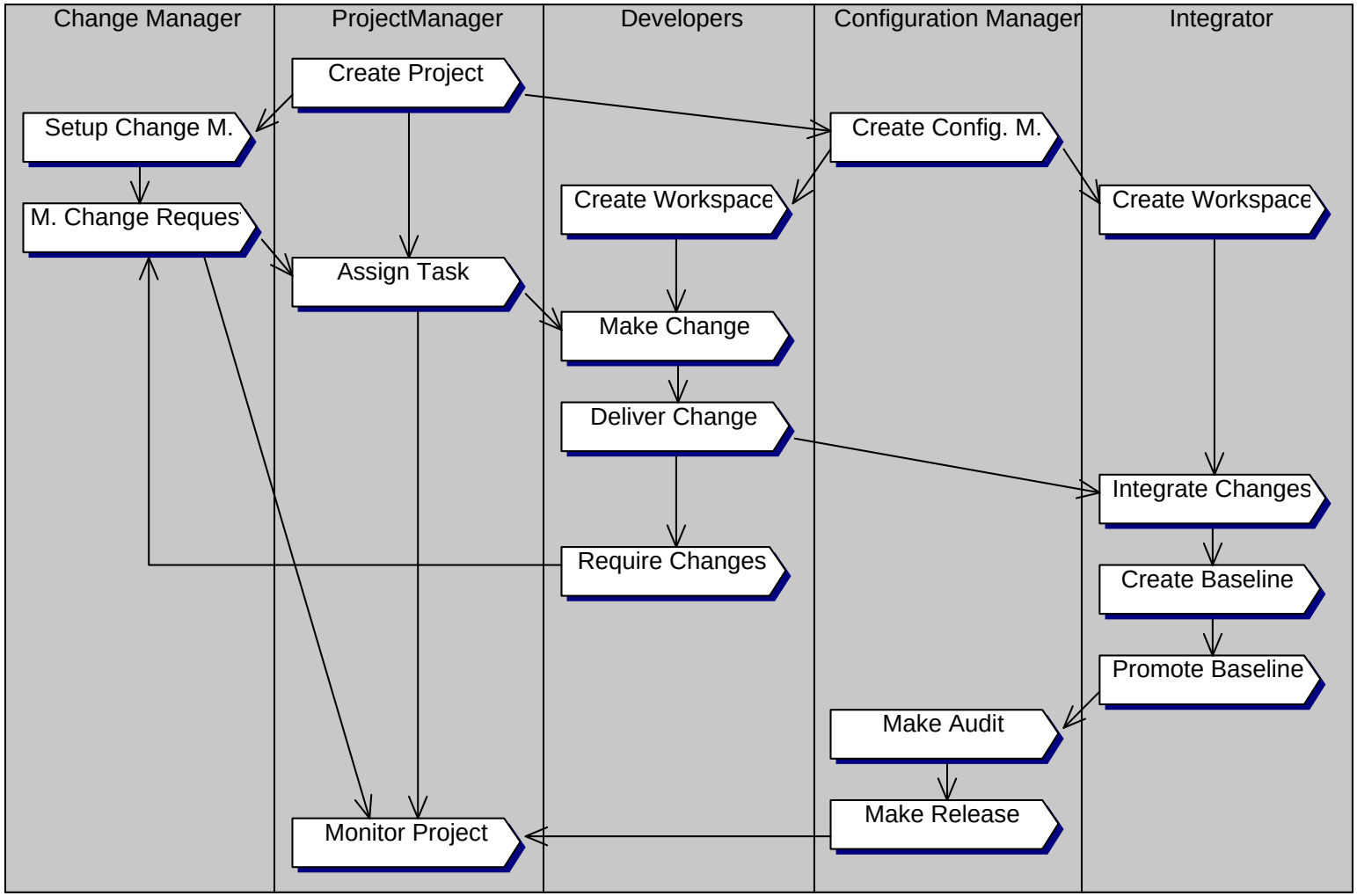
Systementwicklung im V-Modell - zur Erinnerung (siehe SE1):



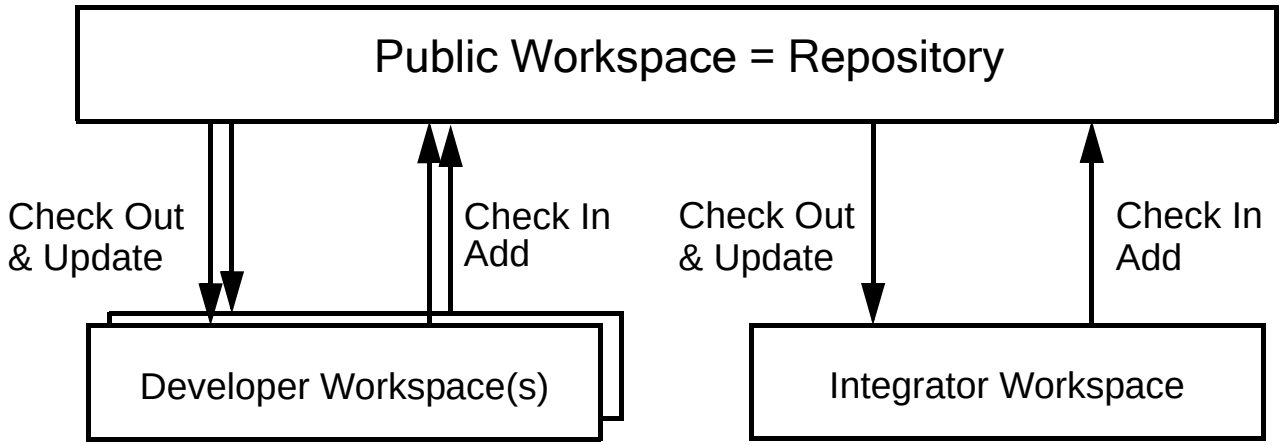
Integration des Konfigurationsmanagements im V-Modell:



Grafische Übersicht über Aufgaben- und Rollenverteilung:



Workspaces für das Konfigurationsmanagement:



- alle Dokumente (Objekte, Komponenten) zu einem bestimmten Projekt werden in einem gemeinsamen Repository (**public workspace**) aufgehoben
- im Repository werden nicht nur aktuelle Versionen, sondern auch alle **früheren Versionen** aller Dokumenten gehalten
- beteiligte Entwickler bearbeiten ihre eigenen Versionen dieser Dokumente in ihrem privaten Arbeitsbereich (private workspace, **developer workspace**)
- es gibt genau einen Integrationsarbeitsbereich (**integrations workspace**) für die Systemintegration

Aktivitäten bei der Arbeit mit Workspaces:

- Personen holen sich Versionen neuer Dokumente, die von anderen Personen erstellt wurden (**checkout**), in ihren privaten Arbeitsbereich.
- Personen passen ihre Privatversionen ggf. von Zeit zu Zeit an neue Versionen im öffentlichen Repository an (**update**).
- Sie fügen (hoffentlich) nur konsistente Dokumente als neue Versionen in das allgemeine Repository ein (**checkin = commit**).
- Ab und an werden neue Dokumente dem Repository hinzugefügt (**add**).
- Jede Person kann für sich entscheiden, ob sie mit neuen oder älteren Versionen im Repository arbeiten will.

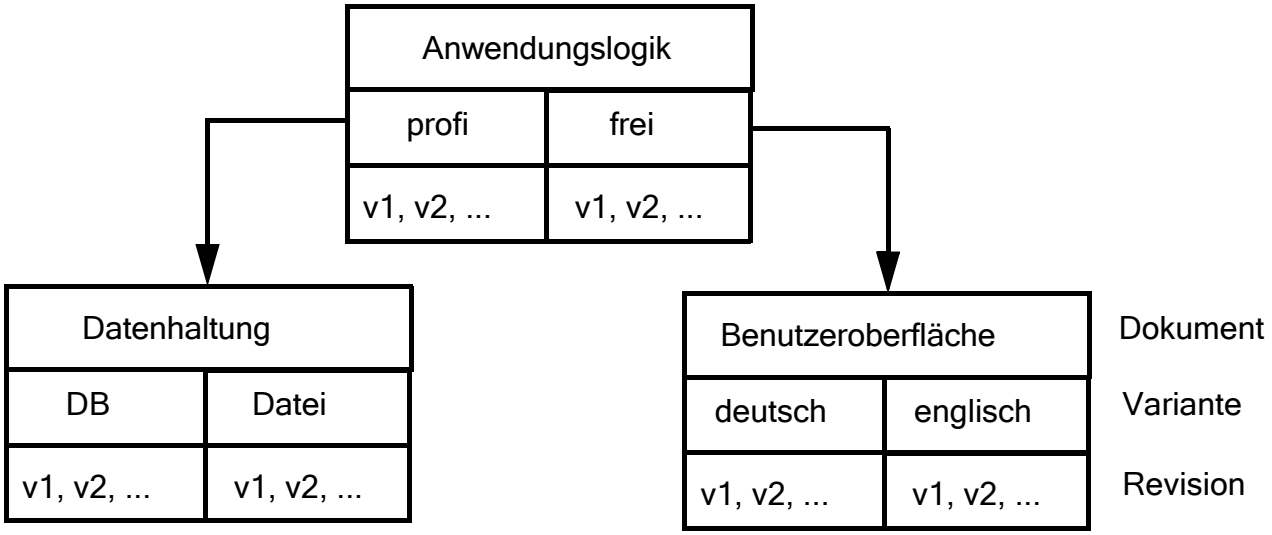
Probleme:

- Wie wird Konsistenz von Gruppen abhängiger Dokumente sichergestellt?
- Was passiert bei gleichzeitigen Änderungswünschen für ein Dokument?
- Wie verwaltet man alle Versionen eines Dokuments effizient?

Weitere Begriffe des Konfigurationsmanagements:

- **Dokument** = Gegenstand, der der Konfigurationsverwaltung unterworfen wird (eine einzelne Datei oder ein ganzer Dateibaum oder ...)
- **(Versions-)Objekt** = Zustand einer Dokument zu einem bestimmten Zeitpunkt in einer bestimmten Ausprägung
- **Varianten** = parallel zueinander (gleichzeitig) existierende Ausprägungen eines Dokuments, die unterschiedliche Anforderungen erfüllen
- **Revisionen** = zeitlich aufeinander folgende Zustände eines Dokuments
- **Konfiguration** = komplexes Versionsobjekt, eine bestimmte Ausprägung eines Programmsystems (oft hierarchisch strukturierte Menge von Dokumenten)
- **Baseline** = eine Konfiguration, die zu einem Meilenstein (Ende einer Entwicklungsphase) gehört und evaluiert (getestet) wird
- **Release** = eine stabile Baseline, die ausgeliefert wird (intern an Entwickler oder extern an bestimmte Kunden oder ...)

Abstraktes Beispiel:



- Anwendungslogik (A), Datenhaltung (D) und Benutzeroberfläche (B) sind drei Pakete (Dokumente), aus denen das Gesamtprodukt besteht
- jedes Paket existiert in zwei Varianten, die jeweils in beliebig vielen zeitlich aufeinander folgenden Revisionen vorliegen können
- es gibt also für das Gesamtprodukt acht mögliche Konfigurationen, falls es für jede Variante genau eine Revision gibt (sonst entsprechend mehr)
- Beispielkonfiguration: { A.profi.v1, D.DB.v2, B.englisch.v2 }

2.2 Versionsmanagement

Versionsmanagement befasst sich (in erster Linie) mit der Verwaltung der zeitlich aufeinander folgenden Revisionen eines Dokuments.

Bekannteste „open source“-Produkte (in zeitlicher Reihenfolge) sind:

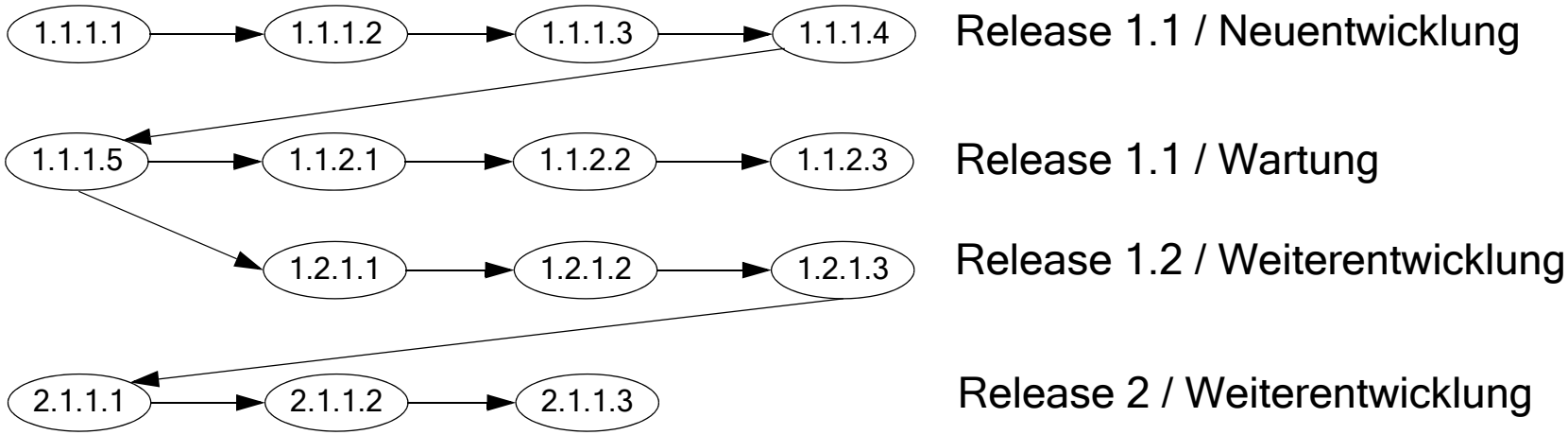
- Source Code Control System **SCCS** von AT&T (Bell Labs):
  - ⇨ effiziente Speicherung von Textdateiversionen als „Patches“
- Revision Control System **RCS** von Berkley/Purdue University
  - ⇨ schnellerer Zugriff auf Textdateiversionen
- Concurrent Version (Control) System **CVS** (zunächst Skripte für RCS)
  - ⇨ Verwaltung von Dateibäumen
  - ⇨ parallele Bearbeitung von Textdateiversionen
- Subversion **SVN** - CVS-Nachfolger von CollabNet initiiert (http://www.collab.net)
  - ⇨ Versionierung von Dateibäumen
- Git, Mercurial, ... als verteilte Versionsmanagementsysteme
  - ⇨ jeder Entwickler hat eigene/lokale Versionsverwaltung

Source Code Control System SCCS von AT&T (Bell Labs):

Je Dokument (Quelltextdatei) gibt es eine eigene **History-Datei,** die alle Revisionen als eine Liste jeweils geänderter (Text-)Blöcke speichert:

- ⇨ jeder Block ist ein **Delta**, das Änderungen zwischen Vorgängerrevision und aktueller Revision beschreibt
- ⇨ jedes Delta hat **SCCS-Identifikationsnummer** der zugehörigen Revision: <ReleaseNo>.<LevelNo>.<BranchNo>.<SequenceNo>

Revisionsbäume von SCCS:



Exkurs zu „diff“ und „patch“ - Beispiel:

```
01:   PROCEDURE P(a, b: INT);
02:      BEGIN
                ...
10:             IF a < b THEN
11:                ...
12:          END
13:      END P;
```

```
01:   PROCEDURE P(v,w: INT);
02:      BEGIN
                ...
10:             IF v < w THEN
11:                ...
12:             ELSE
13:                ...
14:          END
15:   END P;
```

```
01:   PROCEDURE P(v,w: INT);
02:      BEGIN
                ...
10:          WHILE v < w DO
11:             ...
12:          END
13:   END P;
```

diff/patch

diff/patch

```
*** 01,01 *** Hunk 1 ****
!       PROCEDURE P(a, b: INT);
--- 01,01 ---
!       PROCEDURE P(v,w: INT);
***************************
*** 10,11 *** Hunk 2 ****
!               IF a < b THEN
!                  ...
--- 10,13 ---
!               IF v < w THEN
!                  ...
+               ELSE
+                  ...
***************************
```

```
*** 10,13 *** Hunk 1 ****
!               IF v < w THEN
                   ...
-               ELSE
-                  ...
--- 10,11 ---
!               WHILE v < w DO
                   ...
***************************
```

Erläuterungen zu „diff“ und „patch“:

- „**diff**“-Werkzeug bestimmt Unterschiede zwischen (Text-)Dateien = **Deltas**
- ein Delta (diff) zwischen zwei Textdateien besteht aus einer Folge von „**Hunks**“, die jeweils Änderungen eines Zeilenbereiches beschreiben:
  - ⇨ Änderungen von Zeilen: werden mit „**!**“ markiert
  - ⇨ Hinzufügen von Zeilen: werden mit „**+**“ markiert
  - ⇨ Löschen von Zeilen: werden mit „**-**“ markiert
- reale Deltas enthalten unveränderte **Kontextzeilen** zur besseren Identifikation von Änderungsstellen
- ein **Vorwärtsdelta** zwischen zwei Dateien `d1` und `d2` kann als „**patch**“ zur Erzeugung von Datei `d2` auf Datei `d1` angewendet werden
- inverses **Rückwärtsdelta** zwischen zwei Dateien `d1` und `d2` kann als „patch“ zur Wiederherstellung von Datei `d1` auf Datei `d2` angewendet werden
- SCCS-Deltas sind in einer Datei gespeichert, deshalb weder Vorwärts- noch Rückwärts- sondern **Inline-Deltas**

Rückwärtsdeltas - Beispiel:

```
01:   PROCEDURE P(a, b: INT);
02:      BEGIN
                ...
10:             IF a < b THEN
11:                xxx
12:          END
13:      END P;
```

```
01:   PROCEDURE P(v,w: INT);
02:      BEGIN
                ...
10:          IF v < w THEN
11:             xxx
12:          ELSE
13:             yyy
14:          END
15:   END P;
```

```
01:   PROCEDURE P(v,w: INT);
02:      BEGIN
                ...
10:          WHILE v < w DO
11:             xxx
12:          END
13:   END P;
```

diff/patch

diff/patch

```
*** 01,01 *** Hunk 1 ****
!       PROCEDURE P(v,w: INT);
--- 01,01 ---
!       PROCEDURE P(a, b: INT);
***************************
*** 10,13 *** Hunk 2 ****
!               IF v < w THEN
                   xxx
-               ELSE
-                  yyy
--- 10,11 ---
!               IF a < b THEN
                   xxx
***************************
```

```
*** 10,11 *** Hunk 1 ****
!               WHILE v < w DO
                   xxx
--- 10,13 ---
!               IF v < w THEN
                   xxx
+               ELSE
+                  yyy

***************************
```

Genauere Instruktionen zur Erzeugung von Deltas:

Jedes „diff“-Werkzeug hat seine eigenen Heuristiken, wie es möglichst kleine und/oder lesbare Deltas/Patches erzeugt, die die Unterschiede zweier Dateien darstellen. Ein möglicher (und in den Übungen verwendeter) Satz von Regeln zur Erzeugung von Deltas sieht wie folgt aus:

1. Die Anzahl der geänderten, gelöschten und neu erzeugten Zeilen aller Hunks eines **Deltas** zweier Dateien wird möglichst **klein gehalten**.
2. Jeder Hunk beginnt mit genau **einer** unveränderten **Kontextzeile** und enthält sonst nur geänderte, gelöschte oder neu eingefügte Zeilen (Ausnahme: Dateianfang).
3. Aufeinander folgende Hunks sind also durch jeweils **mindestens eine unveränderte Zeile** getrennt.
4. Optional: Anstelle von Löschen und Neuerzeugen einer Zeile *i* verwendet man die **Änderungsmarkierung „!“**

Durch diese Regeln nicht gelöstes Problem:

Wie erkenne ich, ob eine Änderung in Zeile *i* durch Einfügen einer neuen Zeile oder durch Ändern einer alten Zeile zustande gekommen ist?

Beispiel - alte Version (Zeilennummern nicht Bestandteil der Zeile):

1. Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
2. Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile **und enthält**
3. **sonst nur geänderte, gelöschte oder neu eingefügte Zeilen**.
4. Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
5. Änderungsmarkierung.
6. Hunks sind durch mindestens eine unveränderte Zeile getrennt.

Beispiel - neue Version (Zeilennummern nicht Bestandteil der Zeile):

1. Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
2. Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile.
3. Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
4. Änderungsmarkierung.
5. **Aufeinander folgende** Hunks sind durch jeweils mindestens eine unveränderte
6. Zeile getrennt.

Falsches Diff-Ergebnis (5 geänderte Zeilen):

*** 01,06 *** Hunk 1 ***

Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
! Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile und enthält
! sonst nur geänderte, gelöschte oder neu eingefügte Zeilen.
! Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
! Änderungsmarkierung.
! Hunks sind durch mindestens eine unveränderte Zeile getrennt.

--- 01,06 ---

Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
! Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile.
! Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
! Änderungsmarkierung.
! Aufeinander folgende Hunks sind durch mindestens eine unveränderte
! Zeile getrennt.

****

Richtiges Diff-Ergebnis (2 geänderte Zeilen, 1 neue, 1 gelöschte):

```
*** 01,03 *** Hunk 1 ***
   Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
!  Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile und enthält
-  sonst nur geänderte, gelöschte oder neu eingefügte Zeilen.
--- 01,02 ---
   Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
!  Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile.
**********

***05,06 *** Hunk 2 ***
   Änderungsmarkierung.
!  Hunks sind durch mindestens eine unveränderte Zeile getrennt.
--- 04,06 ---
   Änderungsmarkierung.
!  Aufeinander folgende Hunks sind durch mindestens eine unveränderte
+  Zeile getrennt.
**********
```

Beispiel - alte Version (Zeilennummern nicht Bestandteil der Zeile):

1. Die Anzahl der Zeilen aller Hunks eines Deltas **zweier Dateien** wird möglichst
2. klein gehalten.
3. Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile.
4. Hunks sind durch mindestens eine unveränderte Zeile getrennt.
5. Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
6. Änderungsmarkierung.

Beispiel - neue Version (Zeilennummern nicht Bestandteil der Zeile):

1. Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
2. Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile.
3. Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
4. Änderungsmarkierung.
5. **Aufeinander folgende** Hunks sind durch jeweils mindestens eine unveränderte
6. Zeile getrennt.

Darstellung von Deltas im neueren „Unified Diff“-Format:

```
Index: MyFile
===================================================
- - - MyFile (revision 1)
+++ MyFile (revision 2)
@@ -1,1 +1,1 @@
-    PROCEDURE  P(a, b: INT);
+    PROCEDURE  P(v, w: INT);
@@ -10,2 +10,4 @@
-            IF a < b THEN
+            IF v < w THEN
               xxx
+            ELSE
+              yyy
===================================================
```

Es handelt sich „nur“ um eine etwas andere Art der Darstellung von Unterschieden zwischen verschiedenen Dateien (Dateirevisionen): Änderungen werden als Löschung gefolgt von Einfügung dargestellt, ...

Create- und Apply-Patch in Eclipse:

Die „Create Patch“- und „Apply Patch“-Funktionen in Eclipse benutzen genau das gerade eingeführte „Unified Diff“-Format. Dabei werden bei der Erzeugung von Hunks wohl folgende Heuristiken/Regeln verwendet:

- ein Hunk scheint in der Regel mit drei unveränderten Kontextzeilen zu beginnen (inklusive Leerzeilen).
- zwei Blöcke geänderter Zeilen müssen durch mindestens sieben unveränderte Zeilen getrennt sein, damit dafür getrennte Hunks erzeugt werden

Bei der Anwendung von Patches werden folgende Heuristiken/Regeln verwendet:

- werden der Kontext oder die zu löschenden Zeilen eines Patches so nicht gefunden, dann endet die Patch-Anwendung mit einer Fehlermeldung
- befindet sich die zu patchende Stelle eines Textes nicht mehr an der angegebenen Stelle (Zeile), so wird trotzdem der Patch angewendet
- gibt es mehrere (identische) Stellen in einem Text, auf die ein Patch angewendet werden kann, so wird die Stelle verändert, die am nächsten zur alten Position ist

© Prof. Dr. Andy Schürr (TU Darmstadt, FB 18, Institut für Datentechnik)

Eigenschaften von SCCS:

- für beliebige (Text-)Dateien verwendbar (und nur für solche)
- Schreibsperren auf “ausgecheckten” Revisionen
- Revisionsbäume mit manuellem Konsistenthalten von Entwicklungszweigen
- Rekonstruktionszeit von Revisionen steigt linear mit der Anzahl der Revisionen (Durchlauf durch Blockliste)
- Revisionsidentifikation nur durch Nummer und Datum

Offene Probleme:

- ☹ kein Konfigurationsbegriff und kein Variantenbegriff
- ☹ keine Unterstützung zur Verwaltung von Konsistenzbeziehungen zwischen verschiedenen Objekten
- ☹ ...

Probleme mit Schreibsperren:

SCCS realisiert ein sogenanntes „pessimistisches“ Sperrkonzept. Gleichzeitige Bearbeitung einer Datei durch mehrere Personen wird verhindert:

- ⇨ ein Checkout zum Schreiben (**single write access**)
- ⇨ mehrere Checkouts zum Lesen (**multiple read access**)

In der Praxis kommt es aber öfter vor, dass mehrere Entwickler dieselbe Datei zeitgleich verändern müssen (oder Person mit Schreibrecht „commit“ vergisst ...)

Unbefriedigende Lösungen:

- ❑ Entwickler mit Schreibrecht macht „commit“ unfertiger Datei, Entwickler mit dringendstem Änderungswunsch macht „checkout“ mit Schreibrecht
  - ⇨ inkonsistente Zustände in Repository, nur einer darf „Arbeiten“
- ❑ weitere Entwickler mit Schreibwunsch „stehlen“ Datei, machen also „checkout“ mit Leserecht und modifizieren Datei trotzdem
  - ⇨ Problem: Verschmelzen der verschiedenen Änderungen

Revision Control System RCS von Berkley/Purdue University

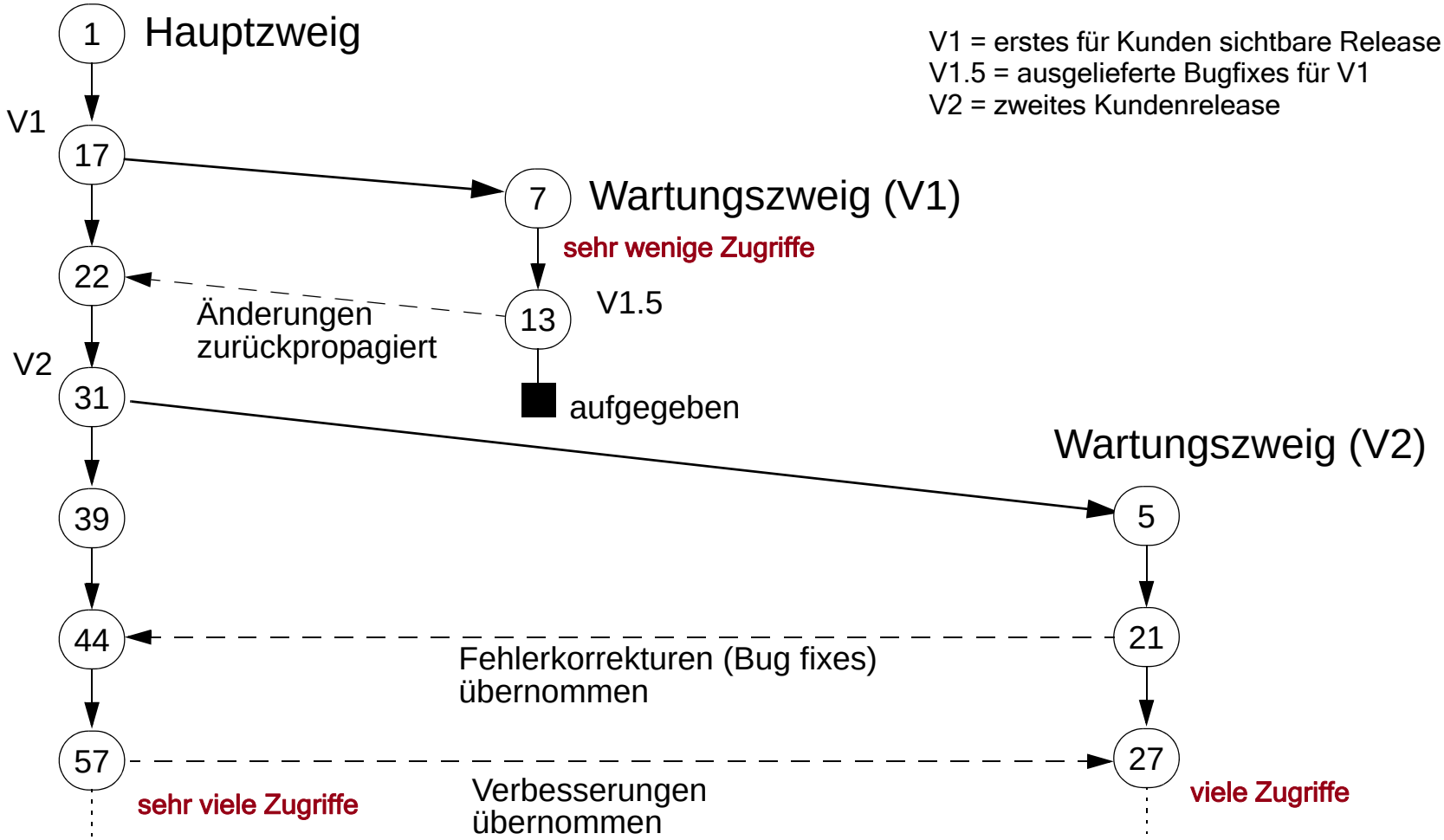
Je Dokument (immer Textdatei) gibt es eine eigene History-Datei, die eine neueste Revision vollständig und andere Revisionen als **Deltas** speichert:

- optionale **Schreibsperren** (verhindern ggf. paralleles Ändern)
- **Revisionsbäume** mit besserem Zugriff auf Revisionen:
  - ⇨ schneller Zugriff auf neueste Revision auf Hauptzweig
  - ⇨ langsamer Zugriff auf ältere Revisionen auf Hauptzweig (mit **Rückwärtsdeltas**)
  - ⇨ langsamer Zugriff auf Revisionen auf Nebenzweigen (mit **Vorwärtsdeltas**).
- Versionsidentifikation auch durch frei wählbare Bezeichner

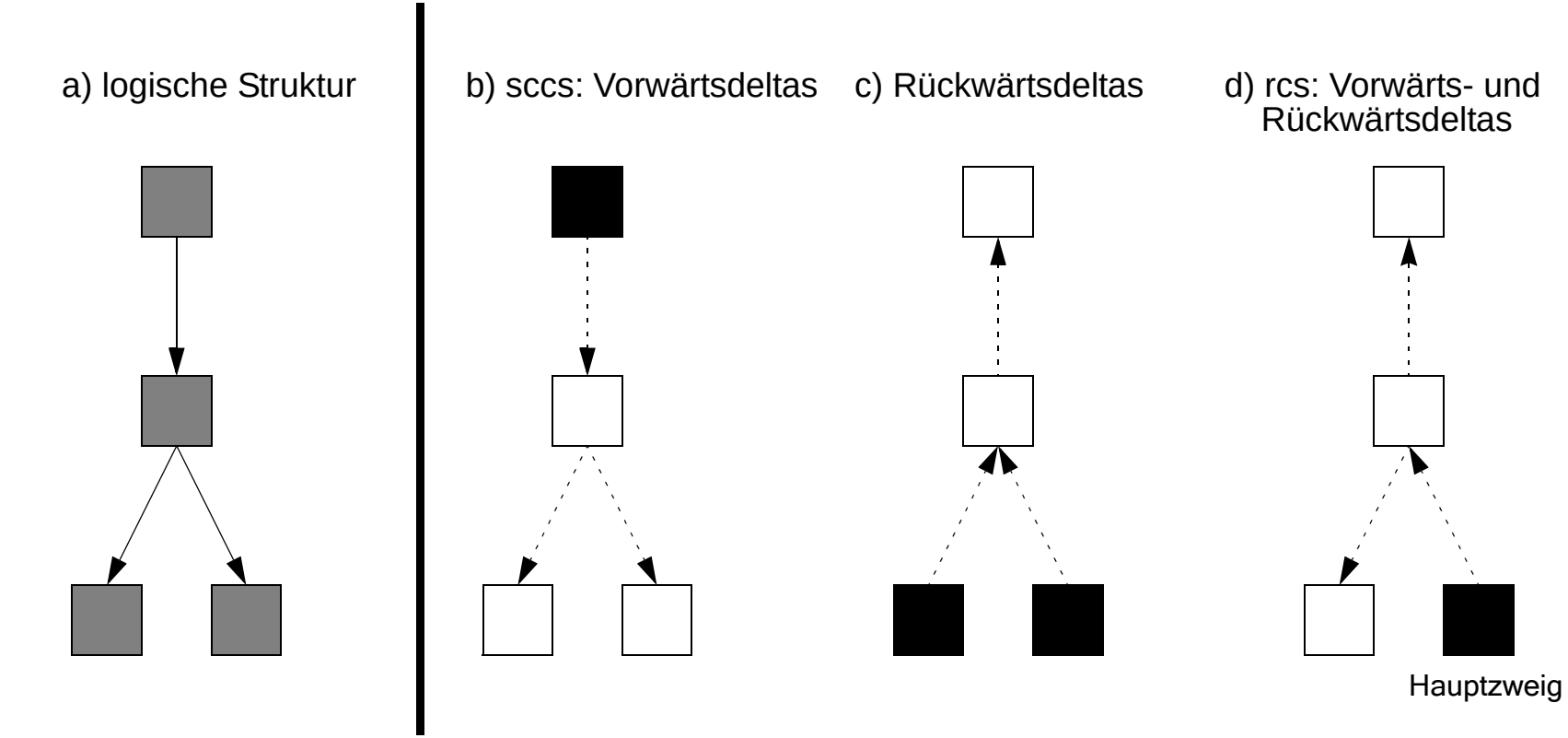
Offene Probleme:

- ☹ kein Konfigurationsbegriff und kein Variantenbegriff
- ☹ ...

Beispielszenario der Softwareentwicklung:



Deltaspeicherung von Revisionen als gerichtete Graphen:



Legende:

: Revision

: direkt verfügbar

: zu rekonstruieren

: Nachfolgerelation

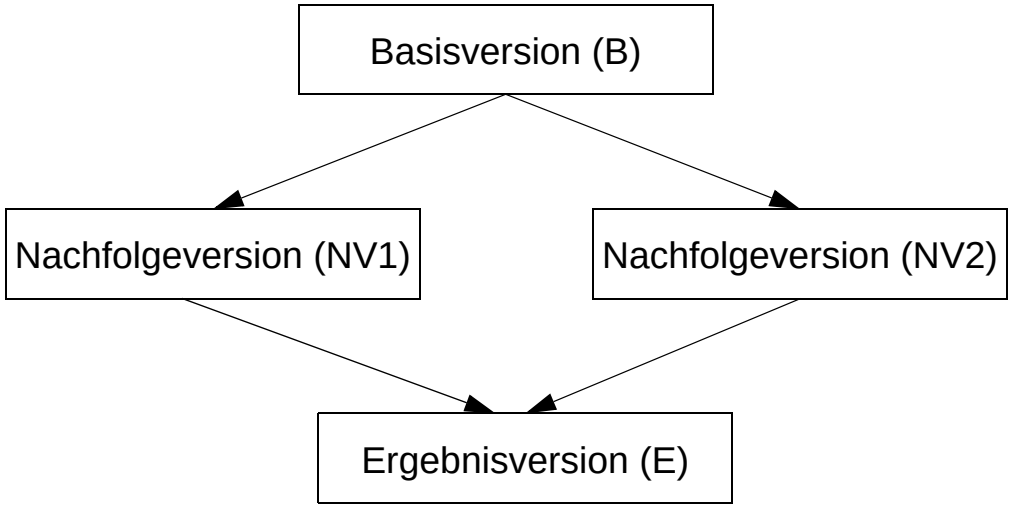
: Rekonstruktionsrelation

Concurrent Version (Management) System CVS:

Zunächst Aufsatz auf RCS (später Reimplementierung), das Revisionsverwaltung für ganze Directorybäume durchführt und zusätzlich anbietet:

- optimistisches Sperrkonzept mit (fast) **automatischem Verschmelzen** (merge) von parallel durchgeführten Änderungen in verschiedenen privaten Arbeitsbereichen
  - ⇨ Dreiwegeverschmelzen mit manueller Konfliktbehebung
- **Revisionsidentifikation** und damit rudimentäres **Releasemanagement** auch durch frei wählbare Bezeichner
  - ⇨ Auszeichnen zusammengehöriger Revisionen durch symbolische Namen (mit Hilfe sogenannter Tags)
- diverse **Hilfsprogramme**
  - ⇨ wie z.B. „cvsann“, das jeder Zeile einer Textdatei die Information voran-stellt, wann sie von wem zum letzten Mal geändert wurde

Dreiwegeverschmelzen von (Text-)Dateien:



Verschmelzungsregeln für Revisionen/Varianten:

- Textzeile in B, NV1 und NV2 gleich ⇒ Textzeile in E
- Textzeile in B aber nicht in NV x oder/und NV y ⇒ Textzeile nicht in E
- Textzeile in NV x oder/und NV y aber nicht in B ⇒ Textzeile in E
- Textzeile aus B in NV1 und NV2 geändert ⇒ manuelle Konfliktbehebung
  (gilt auch für neue Textzeilen in NV1 und NV2 an gleicher Stelle)

Beispiel für das automatische Verschmelzen:

```
PROCEDURE P(a, b: INT);
  BEGIN
     …
     IF a < b THEN
        …
     END
END P;
```

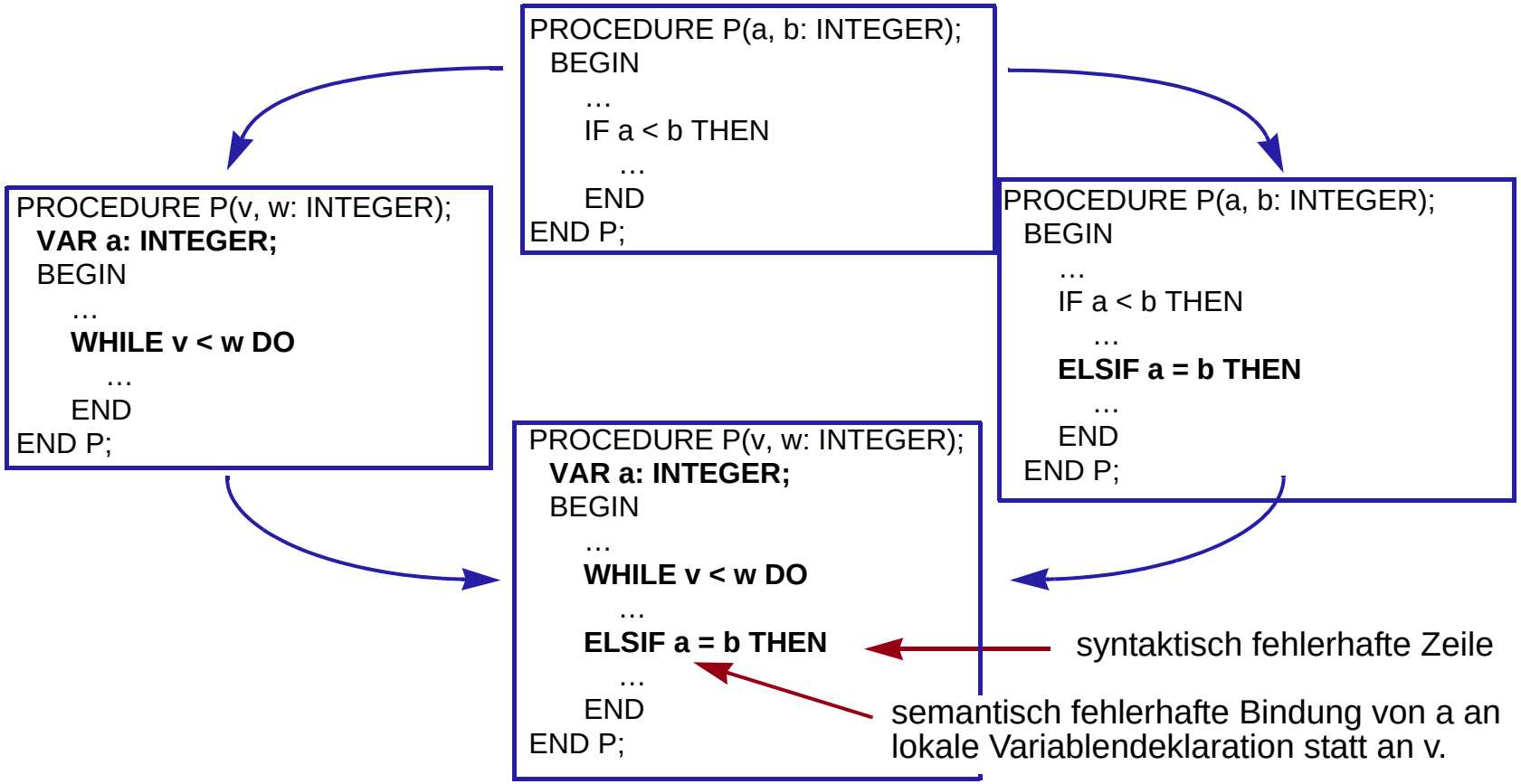
```
PROCEDURE P(v, w: INT);
  BEGIN
     …
     WHILE a < b DO
        …
     END
END P;
```

```
PROCEDURE P(v, w: INT);
     VAR a: INTEGER;
BEGIN
     …
     IF v < a THEN
        …
     ELSIF v = w THEN
        …
     END
END P;
```

```
PROCEDURE P(v, w: INT);
  VAR a: INTEGER;
  BEGIN
     …
!    WHILE a < b DO
!----------------------------
!    IF v < a THEN
        …
     ELSIF v = w THEN
        …
     END
END P;
```

☹ Gemeldeter Konflikt zwischen Änderungen an derselben Zeile (muss vom Benutzer aufgelöst werden)!

Beispiel für Probleme mit dem automatischen Verschmelzen:



☹ Selbst wenn keine Konflikte gemeldet werden, kann Ergebnis syntaktisch oder semantisch fehlerhaft sein (in der Praxis passiert das aber selten)!

Optimistisches Sperrkonzept mit „Merge“:

- Entwickler A macht **checkout** einer Revision n
- Entwickler A verändert Revision n lokal zu n1
- Entwickler B macht **checkout** derselben Revision n
- Entwickler B verändert Revision n lokal zu n2
- Entwickler B macht **commit** seiner geänderten Revision n2
- Entwickler A versucht **commit** seiner geänderten Revision n1
  - ⇨ wird mit Fehlermeldung abgebrochen
- Entwickler A macht **update** seiner geänderten Revision n1
  - ⇨ automatisches **merge** von n1 und n2 mit Basis n führt zu n’
- Entwickler A löst Verschmelzungskonflikte manuell auf und erzeugt n’’
- Entwickler A macht **commit** von Revision n’’ (inklusive Änderungen von B)

Synchronisierung mit Watch/Edit/Unedit:

In manchen Fällen will man wegen größerer Umbauten eine Datei(-Revision) „ungestört“ bearbeiten, also zumindest eine Meldung erhalten, wenn andere Entwickler dieselbe Datei bearbeiten (um sie zu warnen):

- watch on schaltet das Beobachten einer Datei ein; beim checkout wird die gewünschte Revision der Datei nur mit Leserecht lokal zur Verfügung gestellt
- watch off ist das Gegenstück zu watch on
- watch add nimmt Entwickler in Beobachtungsliste für Datei auf, für die er vorher ein checkout gemacht hat (Änderungen an Revisionen der Datei werden per Email allen Entwicklern auf Beobachtungsliste gemeldet)
- watch remove entfernt Entwickler von Beobachtungsliste für Datei
- edit verschafft Entwickler Schreibrecht auf lokal verfügbarer Revision einer Datei und meldet das anderen „interessierten“ Entwicklern (enthält watch add)
- unedit nimmt Schreibrecht zurück und meldet das (enthält watch remove)

Identifikation gewünschter Revisionen - Zusammenfassung:

1. Verwendung von **Revisionsnummern** (Identifikatoren): jede Revision einer Datei erhält eine eigene eindeutige Nummer, über die sie angesprochen wird; Nummern werden nach einem bestimmten Schema erzeugt.

2. Verwendung von **Attributen** (Tags): Revisionen werden durch Attribute und deren Werte indirekt identifiziert; Beispiele sind

   ⇨ Kunde (für den Revision erstellt wurde)

   ⇨ Entwicklungssprache (Java, C, ...)

   ⇨ Entwicklungsstatus (in Bearbeitung, getestet, freigegeben, ...)

   ⇨ ...

3. Verwendung von **Zeitstempeln** (Spezialfall der attributbasierten Identifikation): für jede Revision ist der Zeitpunkt ihrer Fertigstellung (commit) bekannt; deshalb können Revisionen über den Zeitraum ihrer Erstellung (jüngste Revision, vor Mai 2003, etc.) identifiziert werden.

Offene Probleme mit Deltaspeicherung und Verschmelzen:

- Berechnung von **Deltas** funktioniert hervorragend für Textdateien (mit Zeilenenden als „Synchronisationspunkte“ für Vergleich)
- Berechnung minimaler Deltas von Binärdateien wäre wesentlich schwieriger
- Textverarbeitungsprogramme wie Word oder CASE-Tools besitzen deshalb eigene Algorithmen zur Berechnung (und Anzeige) von Deltas
- **Verschmelzung** von Textdateien funktioniert meist gut (kann aber zu tückischen Inkonsistenzen führen)
- Verschmelzung von Binärdateien oder Grafiken/Diagrammen ist im allgemeinen nicht möglich
- Programme wie Word oder CASE-Tools besitzen deshalb eigene Verschmelzungsfunktionen

Verbleibende Mängel von CVS:

- ☹ zugeschnitten auf Textdateien bei (Deltaberechnung und Verschmelzen)
- ☹ keine Versionierung von Verzeichnisstrukturen (Directories)
- ☹ nicht integriert mit „Build-“ und „Changemanagement“
- ☹ keine gute Unterstützung für geographisch verteilte Software-Entwicklung
- ☹ ...

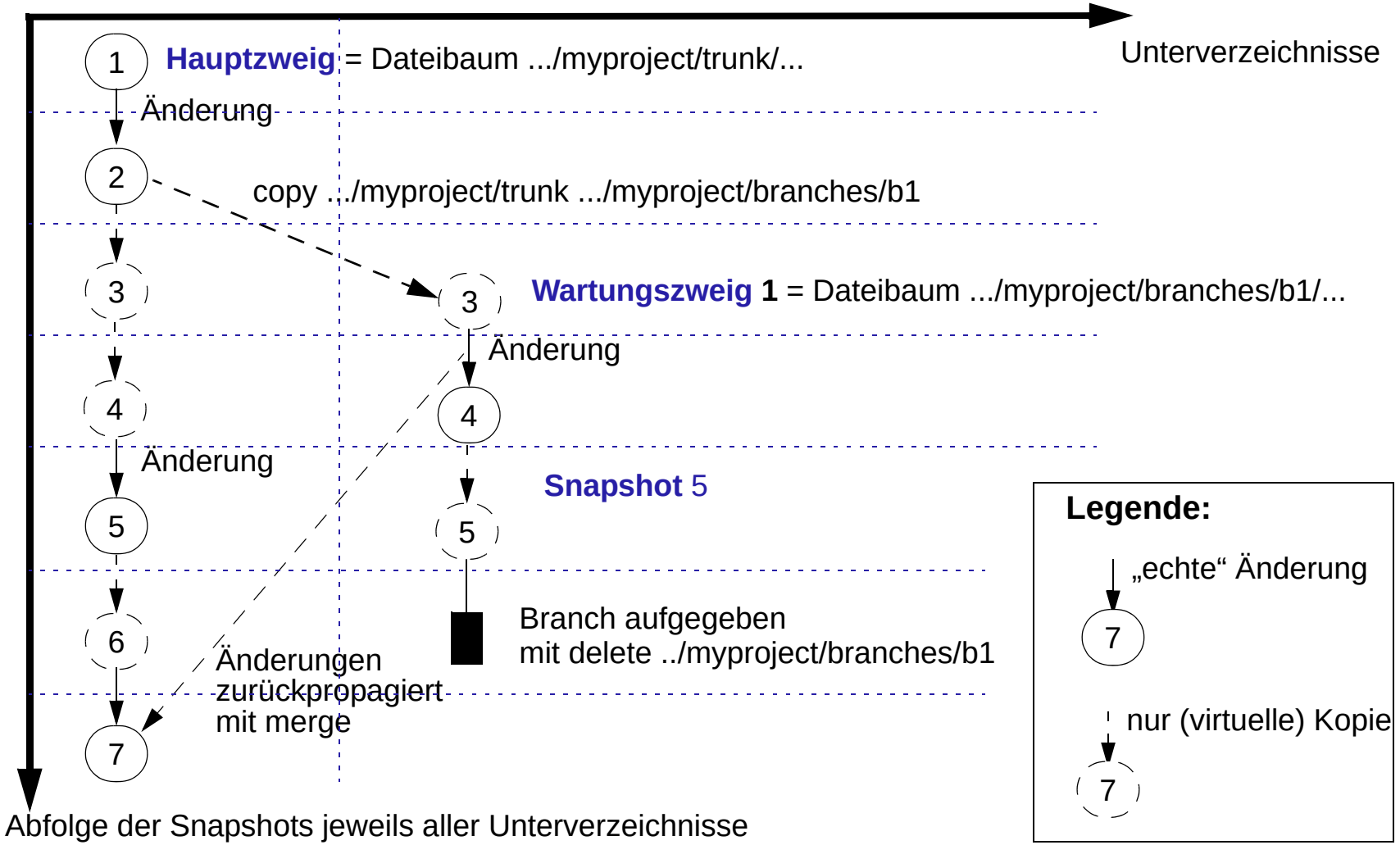
Aber:

- ☺ als „Open Software“ ohne Anschaffungskosten erhältlich
- ☺ immer noch in manchen (Open Software) Projekten eingesetzt
- ☺ Administrationsaufwand hält sich in Grenzen
- ☺ für Build- und Changemanagement gibt es ergänzende Produkte

CVS-Nachfolger Subversion [Wh00], [Po09]:

- ☺ immer ganze Dateibäume werden durch commit versioniert (inklusive Erzeugen, Löschen, Kopieren und Umbenennen von Dateien und Unterverzeichnissen)
- ☺ commit ganzer Verzeichnisse ist eine atomare Aktion (auch bei Server-Abstürzen, Netzwerkzusammenbrüchen, ...), die ganz oder gar nicht erfolgt
- ☺ Metadaten (Dateiattribute, Tags) werden als versionierte Objekte unterstützt
- ☺ Deltaberechnung funktioniert für beliebige Dateien (auch Binärdateien); Verschmelzungsoperationen sind aber weiterhin eher auf Textdateien zugeschnitten
- ☺ geographisch verteilte Softwareentwicklung wird besser unterstützt; Dateibäume und Dateien werden durch URLs identifiziert, Datenaustausch kann über http-Protokoll erfolgen
- ☺ Ungewöhnliche, aber einfache/effiziente Behandlung von mehreren Entwicklungszweigen sowie von Systemrevisionen mit bestimmten Eigenschaften (Tags) als „lazy Copies“ (keine echten Kopien, sondern nur neue Verweise/Links)

Beispielszenario in Subversion:



Einsatz von merge in Subversion:

svn merge -r <snapshot1>:<snapshot2> <source> [<wcpath>]

Dieses Kommando berechnet

- ⇨ alle Unterschiede zwischen <snapshot1> und <snapshot2> (als Vorwärts- bzw. Rückwärtsdelta) des Verzeichnisses <source> mit allen Unterverzeichnissen und Dateien
- ⇨ und wendet die Änderungen auf die Dateien im aktuellen Workspace-Verzeichnis (ggf. angegeben durch <wcpath> = Working Copy Path) an
- ⇨ geändert werden dabei einzelne Dateien und ganze (Unter-)Verzeichnisse

Variante des merge-Kommandos:

svn merge <src1>[@<snapshot1>] <src2>[@<snapshot2>] [<wcpath>]

Dieses Kommando wendet das Delta zweier verschiedener Dateien oder Unterverzeichnisse vorgegebener Revisionen (durch Snapshot-Nummer) auf den aktuellen Workspace an. Wird keine Revision angegeben, so wird jeweils die neueste Revision (Head) verwendet.

Beispiele für Verwendung von merge:

- Anwendung aus Sicht von Wartungszweig 1 (propagate changes):

  merge -r 3:4 .../myproject/branches/b1 [wcpath]

  ⇨ propagiert alle Änderungen von Revision 3 auf Revision 4 des Wartungszweiges in die ausgecheckte „Working Copy“ des Hauptzweiges oder eines anderen Wartungszweiges

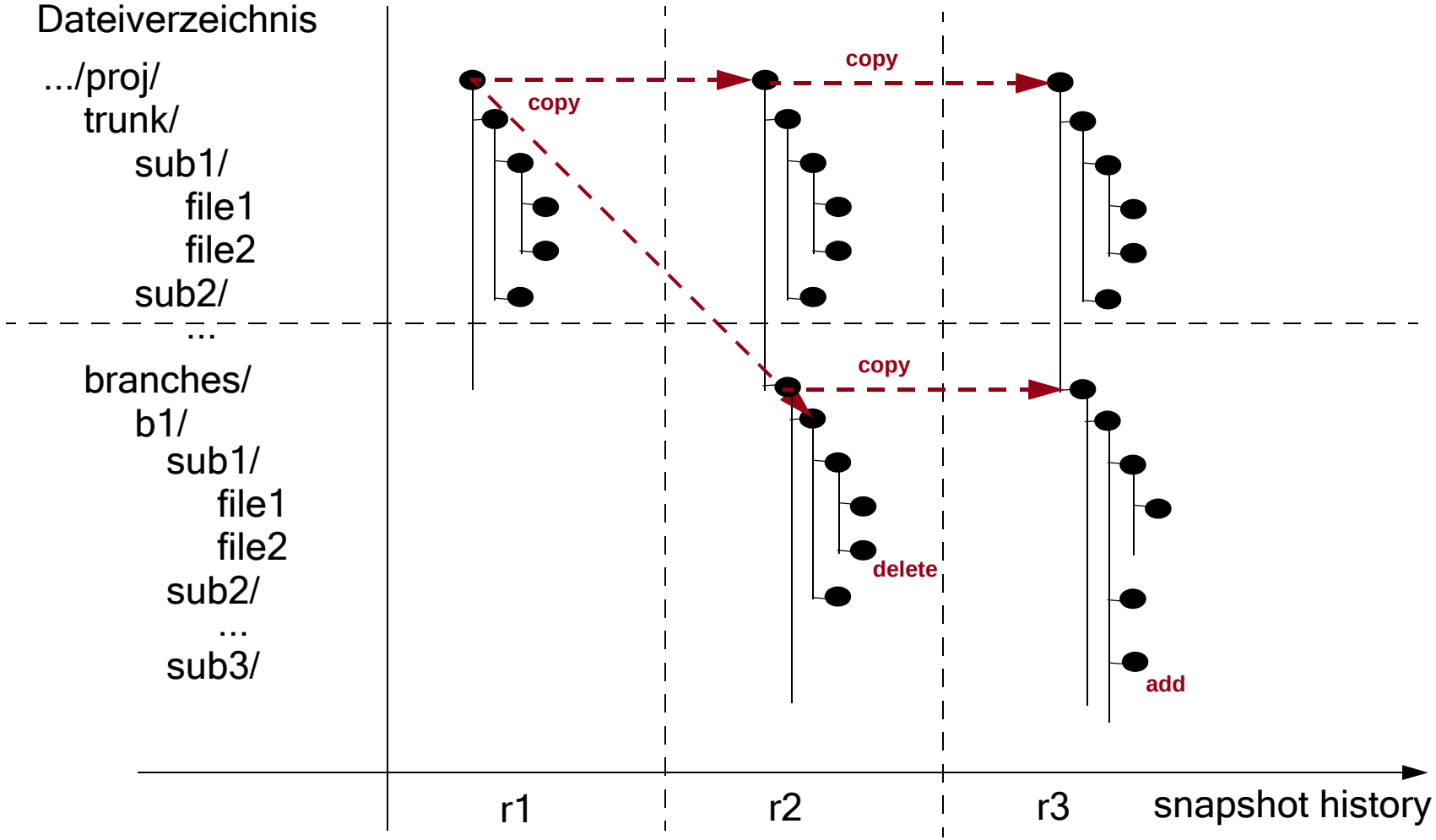
  ⇨ die neue Revision des Hauptzweiges kann anschließend als Snapshot 7 „eingecheckt“ werden

- Anwendung aus Sicht von Wartungszweig 1 (undo changes):

  merge -r 4:3 .../myproject/branches/b1 [wcpath]

  ⇨ eliminiert alle propagierten Änderungen aus Wartungszweig 1 wieder durch Berechnung und Anwendung des inversen Deltas

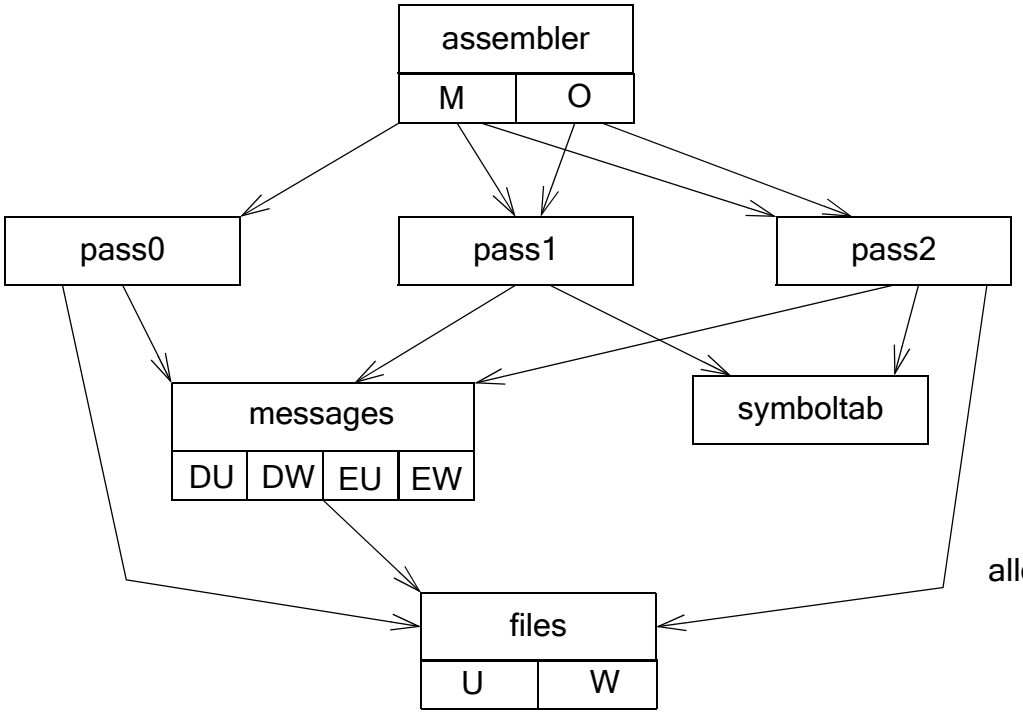
Zweidimensionaler Zustandsraum des Subversion-Repositories:



2.3 Variantenmanagement (Software-Produktlinien)

Das Variantenmanagement befasst sich mit der Verwaltung neben-einander existierender Versionen eines Dokuments, die jeweils eine zeitliche Entwicklungsgeschichte besitzen.

Beispiel:



Legende:

M = mit Makros
O = ohne Makros
D = deutsche Meldungen
E = englische Meldungen
U = für Unix
W = für Microsoft Windows

Modul

Importbeziehung

Denkbare Varianten:

(M,D,U) = mit Makros für Deutsch und Unix
(O,D,U)= ohne Makros für Deutsch, Unix
(M,E,U) = ...

alle Module/Komponenten hängen davon ab

global

Erläuterungen zum Variantenbeispiel (SW-Produktlinie):

- es handelt sich um die Softwarearchitektur eines fiktiven Assemblers, die aus 8 Moduln (Pakete, Komponenten, ...) besteht und in 6 Varianten existiert
- es gibt ein Hauptprogramm assembler, das ein Assemblerprogramm in 2 oder 3 Durchläufen in Maschinencode übersetzt (ohne und mit Behandlung von Makros)
- pass0 expandiert Makros in dem Assemblerprogramm
  (liest Eingabe aus Datei, schreibt Ausgabe auf Datei)
- pass1 merkt sich Sprungmarken, Konstanten und deren Werte in einer Symboltabelle
- pass2 führt die eigentliche Übersetzung mit Hilfe der Symboltabelle durch
- messages bietet eine Abbildung von Fehlernummern auf Texte in Dateien
  (Implementierung hängt in kleinen Teilen von Sprache und Betriebssystem ab)
- files bildet die Schnittstelle zum Dateisystem (für Unix und Windows)
- globals sind Konstanten, Funktionen etc., die überall benötigt werden

Software-Produktlinien-Entwicklung = „Variantenmanagement im Großen“:

Eine **Software-Produkt-Linie** (SPL) ist eine Menge „verwandter“ Softwaresysteme

- für eine bestimmte **Anwendungsdomäne**
- die auf Basis einer gemeinsamen **Plattform** (Rahmenwerk)

entwickelt werden. Die Plattform enthält alle SW-Artefakte, die für alle Instanzen der Produktlinie gleich sind. Damit ist die Software-Produktlinien-Entwicklung (SPLE = Software Product Line Engineering) eine logische Verallgemeinerung des Variantenmanagements eines Softwaresystems.

Man unterscheidet beim SPLE zwischen

- der **Domänenentwicklung** (Domain Engineering) der gemeinsamen Plattform (development **for** reuse)
- der **Anwendungsentwicklung** (Application Engineering) von SPL-Instanzen (development **with** reuse)

Domänen- und Anwendungsentwicklung:

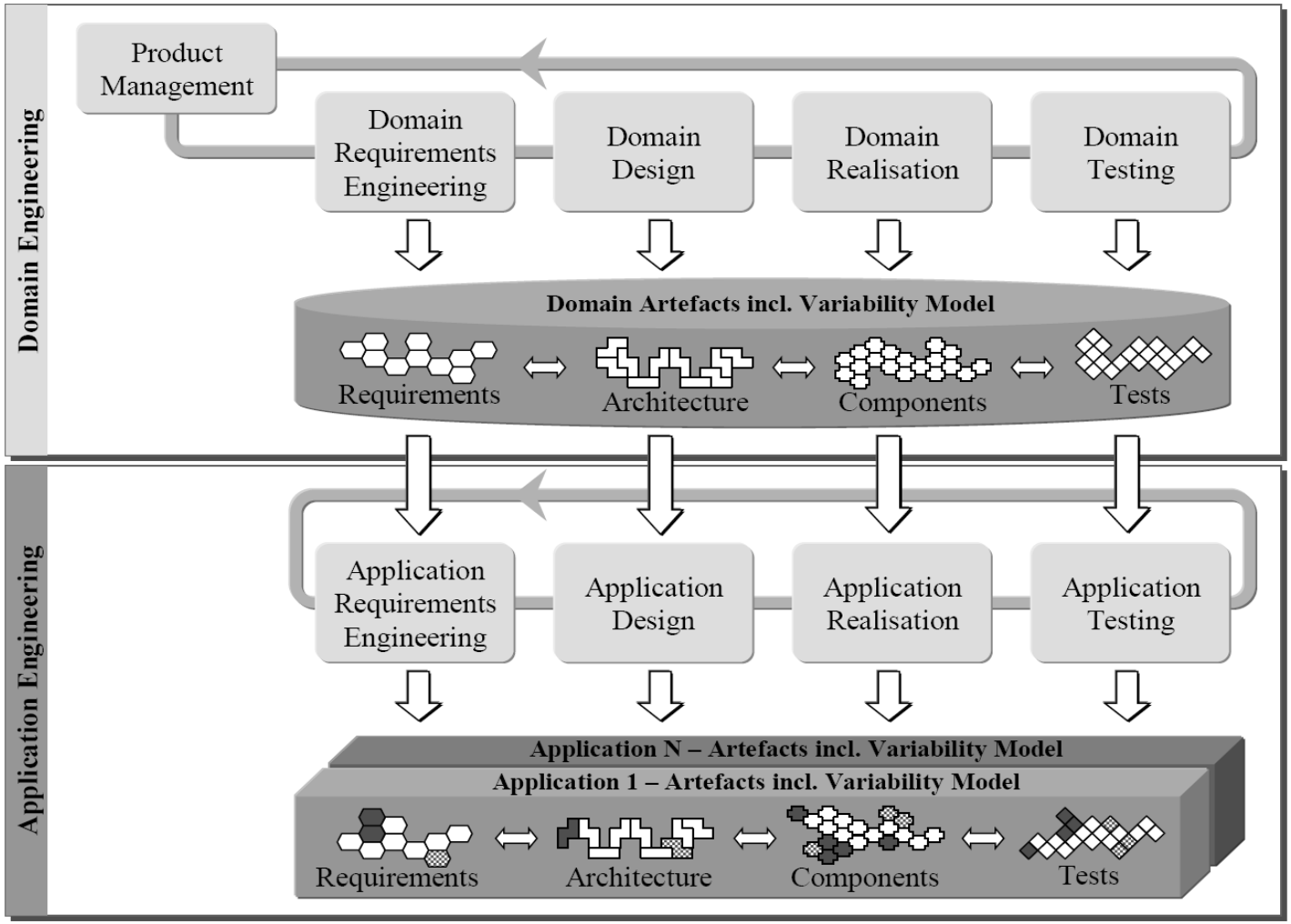


Abb. aus [PBL05]

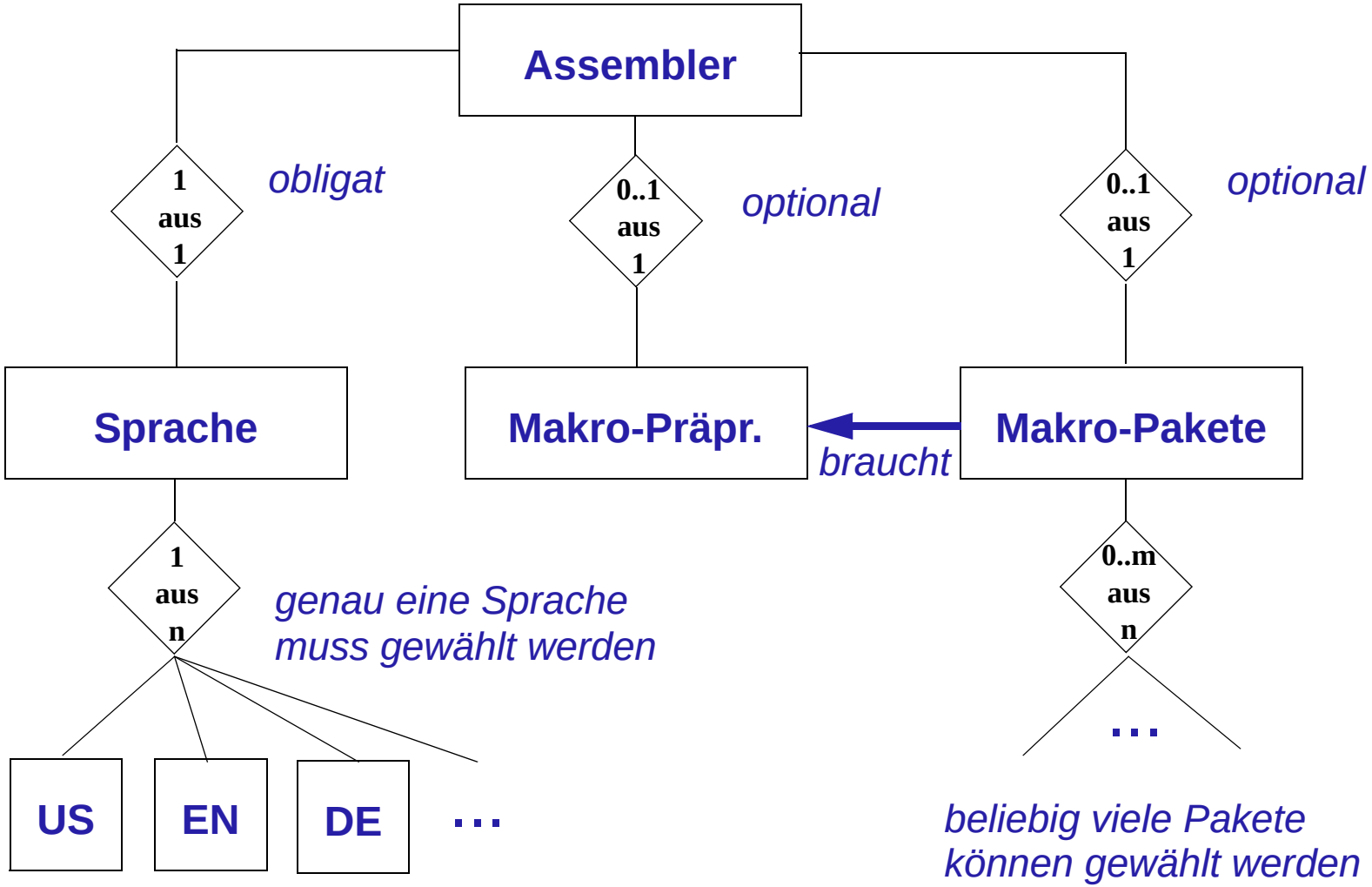
Variabilitätsmodell (Feature-Modell) einer SPL:

So genannte Variabilitäts- oder Feature-Modelle beschreiben alle möglichen Eigenschaften einer Software-Produktlinie und zulässige Kombinationen (Konfigurationen, Varianten, Instanzen) dieser Produktlinie.

Eine Vielzahl von Notationen und Werkzeugen unterstützen die Erstellung solcher Modelle durch

- Festlegung aller auswählbaren Eigenschaften/Merkmale = **Features** der Instanzen der Produktlinie (vor allem die optionalen/alternativen Eigenschaften)
- **hierarchische Zerlegung** von Merkmalen in Untermerkmale
  (das Feature „Dialog-Sprache = Englisch“ wird zerlegt in „US-Englisch“, ...)
- Festlegung von **Auswahl-Optionen** der Art eins-aus-n, m-aus-n, ...
  (eine SPL-Instanz läuft entweder auf Windows oder auf Unix)
- Definition von **Abhängigkeiten** und Ausschlusskriterien einzelner Merkmale in ganz verschiedenen Teilhierarchien

Beispiel eines Variabilitätsmodells (Assembler):



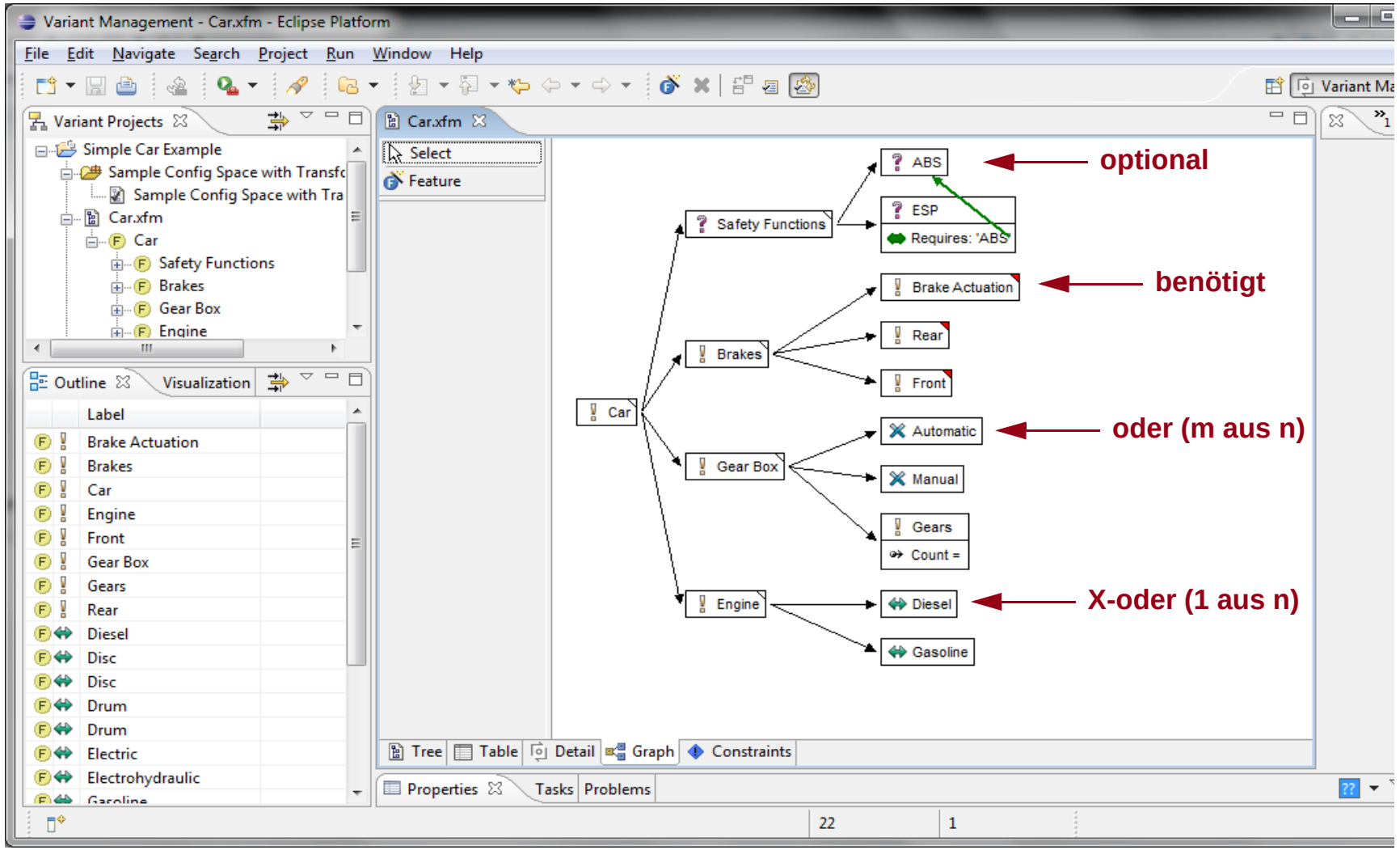
Werkzeugunterstützung für SPL-Entwicklung:

- (grafische) Erstellung von Variabilitätsmodellen
- Unterstützung bei der Auswahl erlaubter Merkmalkombinationen
- Erzeugung von Produktinstanzen (Implementierungen) für bestimmte Merkmalkombinationen
- [Unterstützung beim systematischen Test von SPLs]
- ...

Beispiel-Werkzeug:

Weit verbreitet (zumindest im deutschen Sprachraum) ist das Werkzeug pure::variants der Firma pure systems (siehe http://www.pure-systems.com/). Es benutzt eine andere grafische Darstellung von „Featuremodellen“ als das vorgestellte Variabilitätsmodell (Anmerkung: es gibt dutzende verschiedene Notationen, die alle mehr oder weniger äquivalent von ihrer Ausdruckskraft her sind).

Das Produktlinienverwaltungswerkzeug pure::variants:



Variantenmanagement für Quelltextdateien:

Hierfür gibt es kaum eigene Unterstützung. Im wesentlichen bleiben folgende Möglichkeiten zur Verwaltung verschiedener Varianten eines Dokuments:

- Mit Hilfe der **Versionsverwaltung** wird je Variante ein eigener Entwicklungszweig gepflegt (mit entsprechendem Tag). Änderungen von einem Zweig werden mit Hilfe der Dreiwegeverschmelzung in andere Zweige propagiert.
- Die verschiedenen Varianten werden alle in einer Datei gespeichert; durch **Bedingungsmakros** (#ifdef Unix ...) werden die für eine Variante benötigten Quelltextteile ein- und ausgeblendet (siehe pure::variants)
- Die verschiedenen Varianten einer Klassenimplementierung werden in Unterklassen ausgelagert (Einsatz von **Vererbung**)
- Benötigte Varianten werden (aus einer domänenspezifischen Beschreibung) **generiert** (mit Quelltexttransformationen, aspektorientierter Programmierung, ...)
- **Achtung**: Auswahl/Konfiguration/Transformation benötigter Quelltextdateien durch das Build-Management (siehe Abschnitt 2.5)

2.4 Releasemanagement

Ein Release ist eine an Kunden ausgelieferte Konfiguration eines (Software-)Systems bestehend aus ausführbaren Programmen, Bibliotheken, Dokumentation, Quelltexte, Installationsskripte, ...

Das Releasemangement dokumentiert ausgelieferte Konfigurationen und stellt deren Rekonstruierbarkeit sicher.

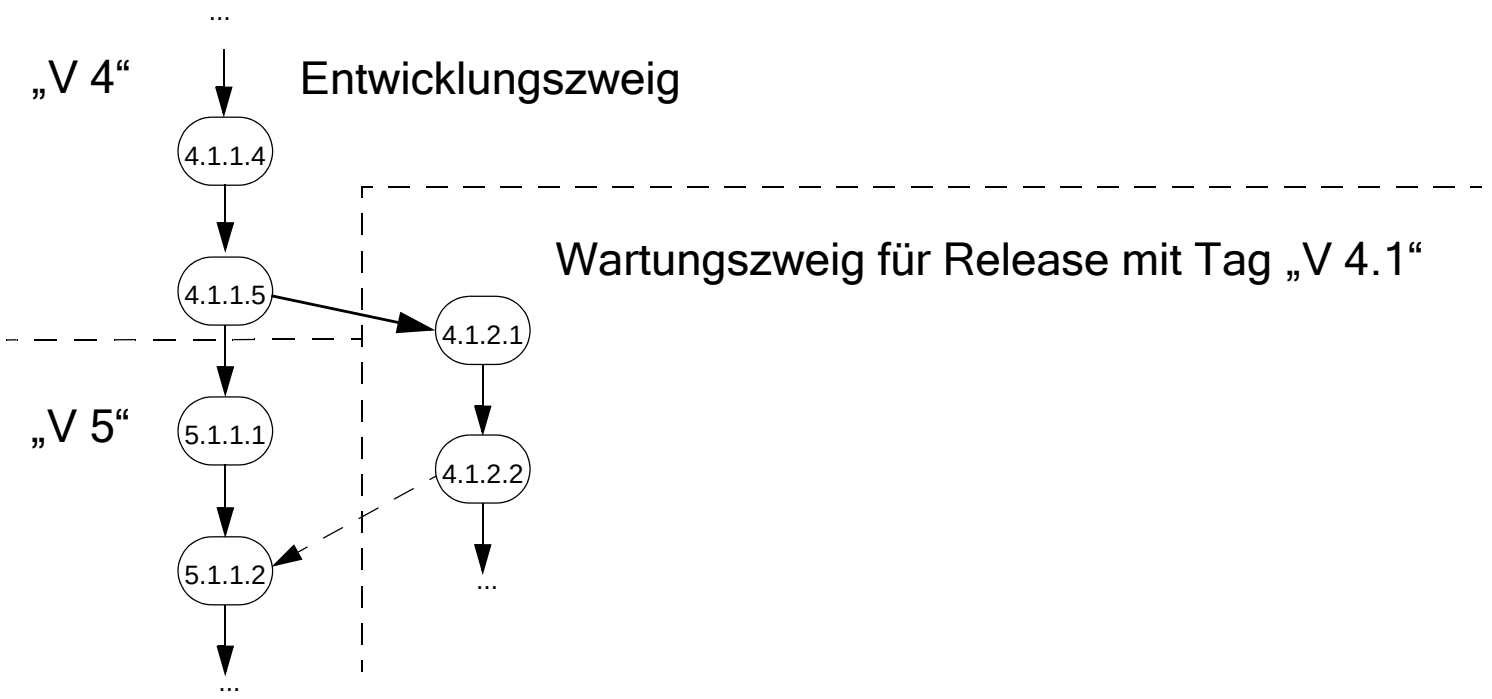
Aufgaben des Releasemanagement:

- Festlegung der (zusätzlichen) Funktionalität eines neuen Releases
- Festlegung des Zeitpunktes der Freigabe eines neuen Releases
- Erstellung und Verbreitung eines Releases (siehe auch Buildmanagement)
- Dokumentation des Releases:
  - ⇨ welche Revisionen welcher Dateien sind Bestandteil des Releases
  - ⇨ welche Compilerversion wurde verwendet
  - ⇨ Betriebssystemversionen auf Entwicklungs- und Zielplattform

Planungsprozess für neues Release:

1. Vorbedingungen für neues Release werden überprüft:
   - ⇨ viel Zeit vergangen (neues Release aus Publicity-Gründen)
   - ⇨ viele Fehler behoben (neues Release mit allen „Patches“)
   - ⇨ viele neue Funktionen hinzugefügt
2. Weiterentwicklung wird eingefroren (freeze):
   - ⇨ **Feature Freeze** (Soft Freeze): nur noch Fehlerkorrekturen und kleine Verbesserungen erlaubt (nur vermutlich nicht destabilisierende Änderungen)
   - ⇨ **Code Freeze** (Hard Freeze): nur noch absolut notwendige Änderungen, selbst „gefährliche“ Fehlerkorrekturen werden verboten
3. Letzte Fehlerkorrekturen und umfangreiche Qualitätssicherungsmaßnahmen (Tests) werden durchgeführt
4. Release wird freigegeben und weiterverbreitet:
   - ⇨ Weiterentwicklung (neuer Releases) wird wieder aufgenommen
   - ⇨ freigegebenes Release muss parallel dazu gepflegt werden

Standardlösung für parallele Pflege und Weiterentwicklung:

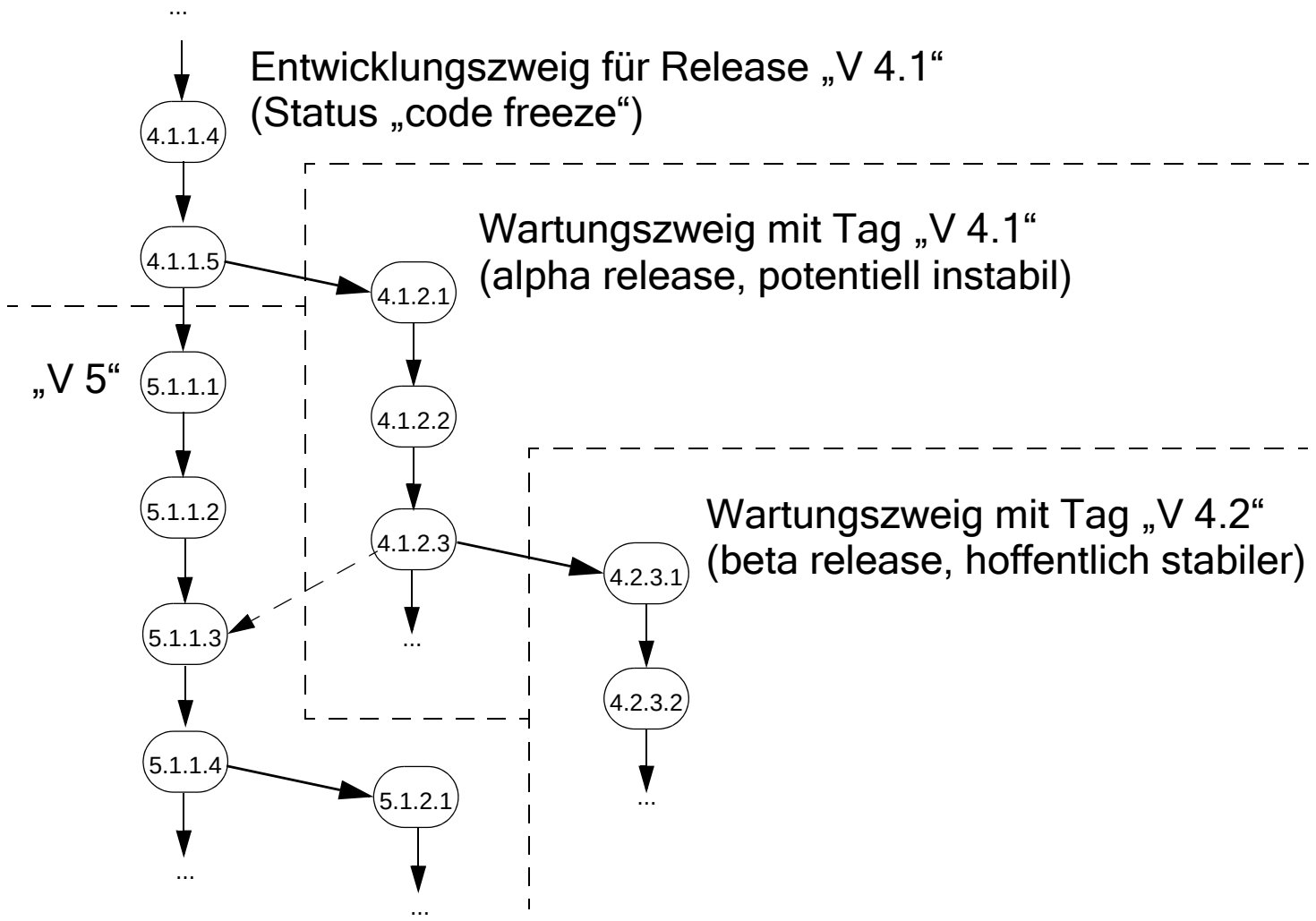


- für Weiterentwicklung des nächsten Releases und Wartung des gerade freigegebenen Releases werden unterschiedliche Revisionszweige verwendet
- alle auf dem Wartungszweig liegenden Revisionen erhalten den Namen des Releases (Versionsnummer) als Tag

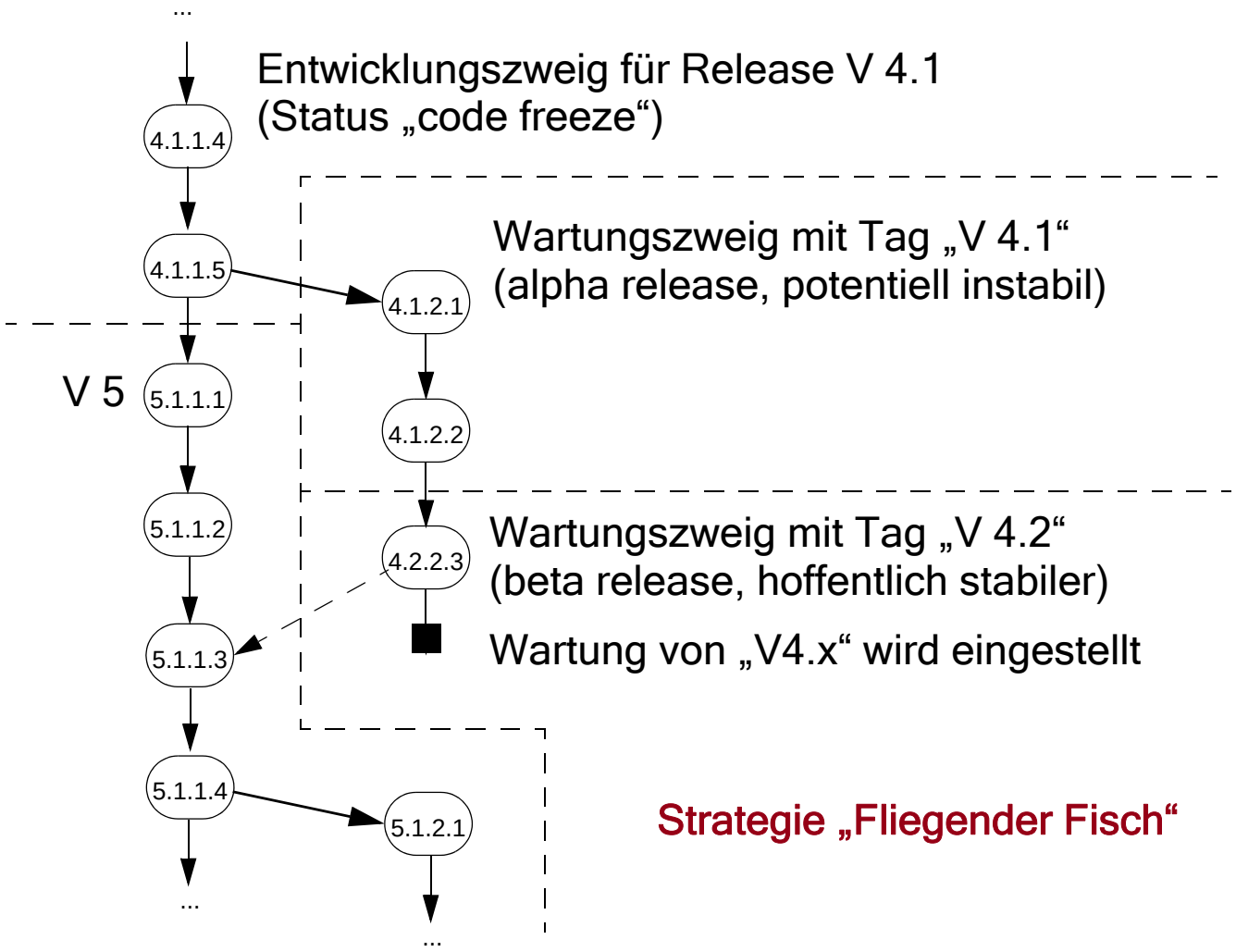
Strategien für die Vergabe von Versions- und Revisionsnummern:

- Software-Releases erhalten üblicherweise zweistellige **Versionsnummern**
- die erste Stelle wird erhöht, wenn sich die Funktionalität signifikant ändert
- die zweite Stelle wird für kleinere Verbesserungen erhöht
- oft wird erwartet, dass x.2, x.4, ... stabiler als x.1, x.3, ... sind
- die (in cvs vierstelligen) internen **Revisionsnummern** eines KM-Systems müssen mit den extern sichtbaren Versionsnummern nichts zu tun haben
- in den vorigen Beispielen galt aber:
  Versionsnummer = ersten beiden Stellen der Revisionsnummer
- oft werden jedoch die ersten beiden Stellen der Revisionsnummern (in cvs) nicht verwendet und bleiben auf „1.1“ gesetzt

Hauptentwicklungszweig und mehrere Wartungsnebenzweige:

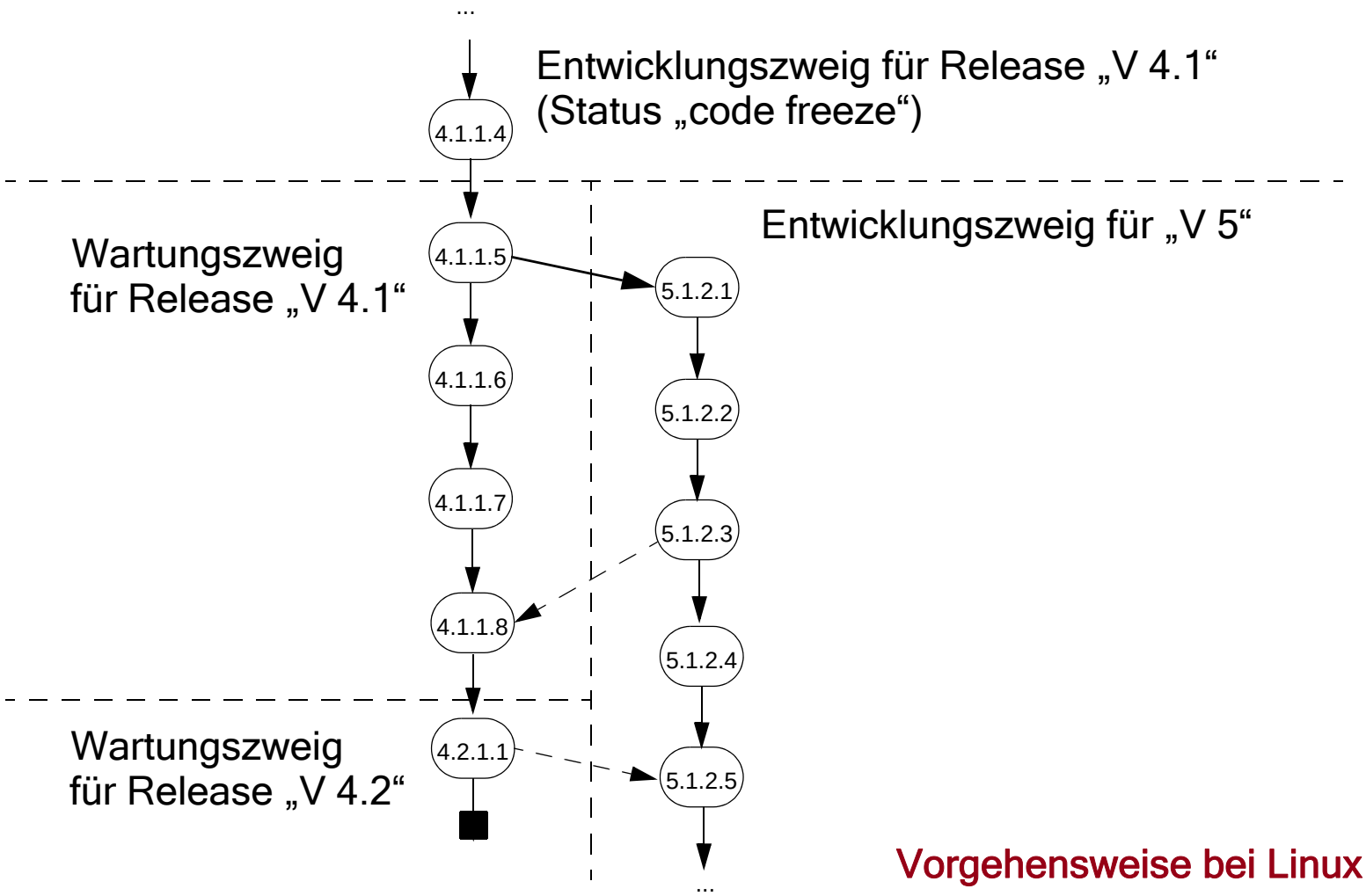


Hauptentwicklungszweig und ein Wartungsnebenzweig:



Strategie „Fliegender Fisch“

Inversion von Wartungs- und Entwicklungszweig:

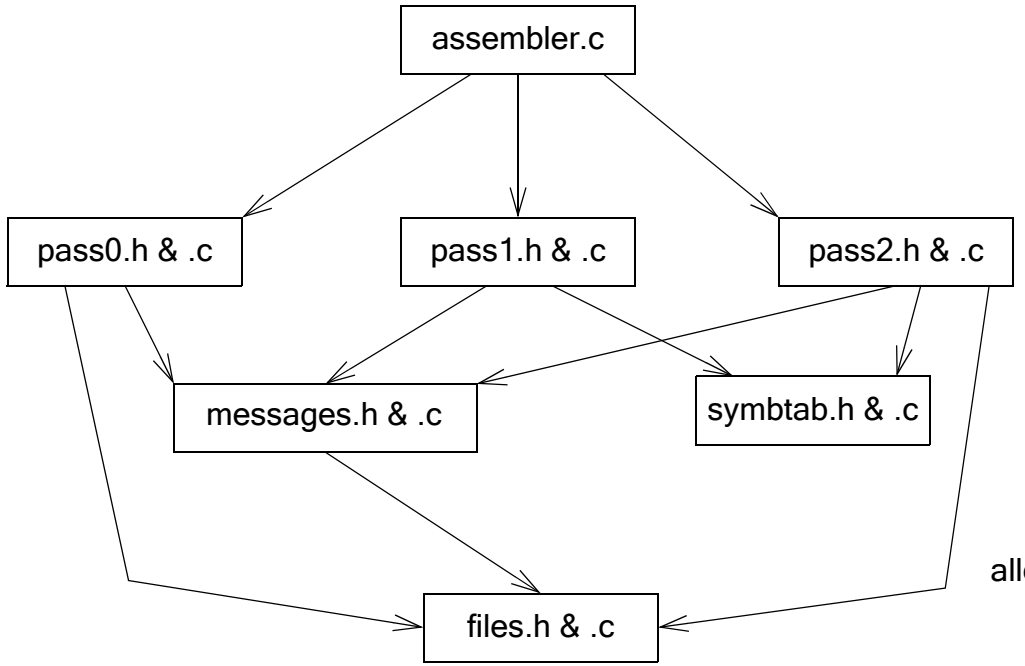


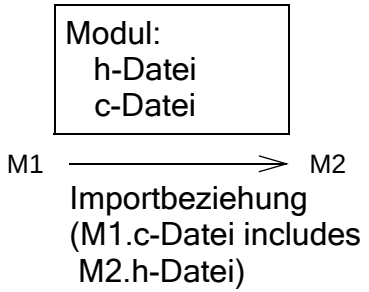
Vorgehensweise bei Linux

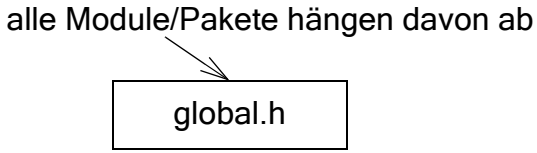
2.5 Buildmanagement

Das Buildmanagement (Software Manufacturing) automatisiert den Erzeugungsprozess von Programmen (Software Releases).

Altes Beispiel - einfache Assemblerimplementierung in C:



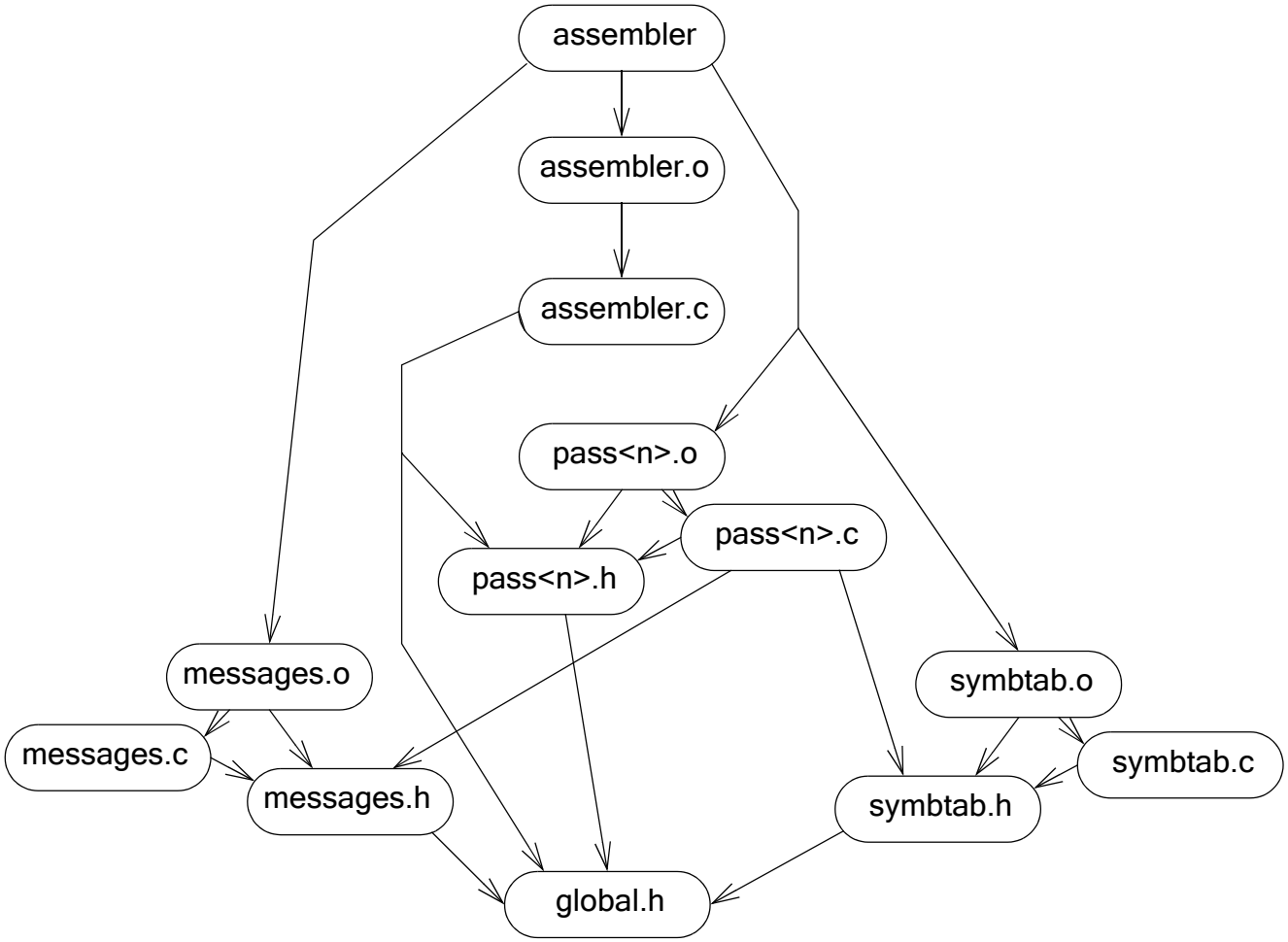




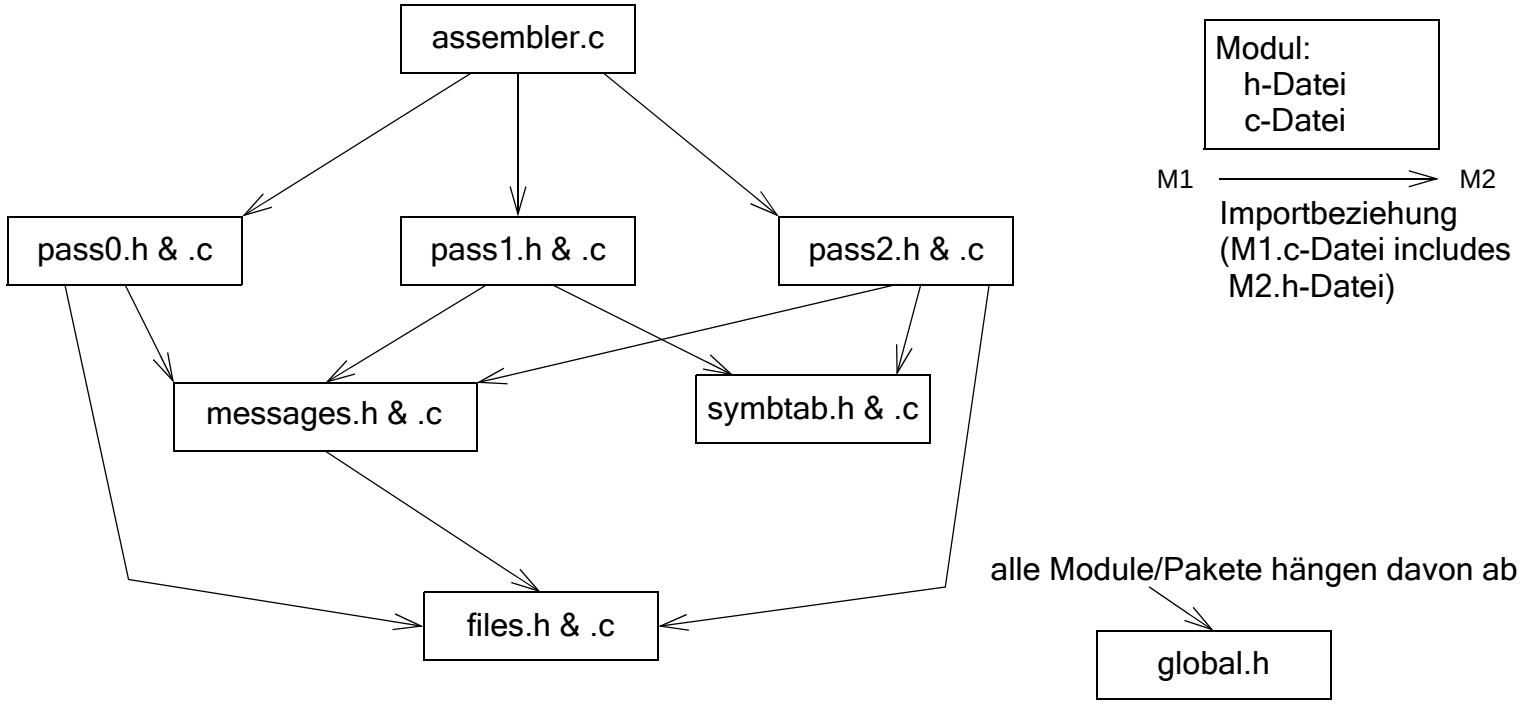
Erläuterungen zum Beispiel - Implementierung in C:

- jedes „normale“ Modul M besteht aus zwei Textdateien:
  - ⇨ die **Header-Datei** M.h beschreibt die Schnittstelle des Moduls, die von anderen Moduln benötigt wird (Konstanten, Typen, Prozedurdefinitionen)
  - ⇨ die **Implementierungsdatei** M.c enthält die C-Quelltexte der Prozeduren
- das Modul global besteht nur aus einer Textdatei global.h:
  - ⇨ die Datei enthält überall benötigte Konstanten- und Typdefinitionen
  - ⇨ „überall“ = in allen anderen .h- und .c-Dateien
- das Hauptprogramm assembler besitzt keine Schnittstelle für andere Module; es besteht also nur aus einer .c-Datei
- jede Quelltextdatei M.c mit **Suffix .c** wird in eine Objektdatei M.o übersetzt; dabei werden .h-Dateien aller importierten Module verwendet
- alle Objektdateien mit **Suffix .o** werden zu einem ausführbaren Programm zusammengebunden

Altes Beispiel - genauerer Abhängigkeitsgraph:

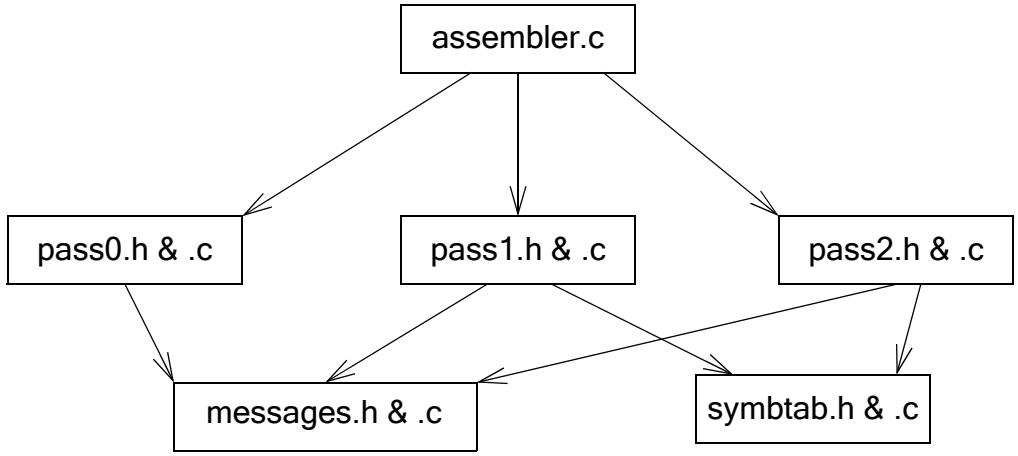


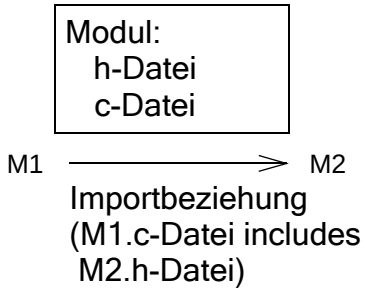
Übersetzungs- und Bindevorgänge - Beispiel 1:



1. die Implementierung einer Prozedur von messages = messages.c ändert sich
2. nur messages.c muss neu in messages.o übersetzt werden: cc -c messages.c
3. Hauptprogramm assembler muss neu aus allen o-Dateien erzeugt werden:
   cc -o assemb assemb.o pass0.o pass1.o pass2.o messages.o files.o

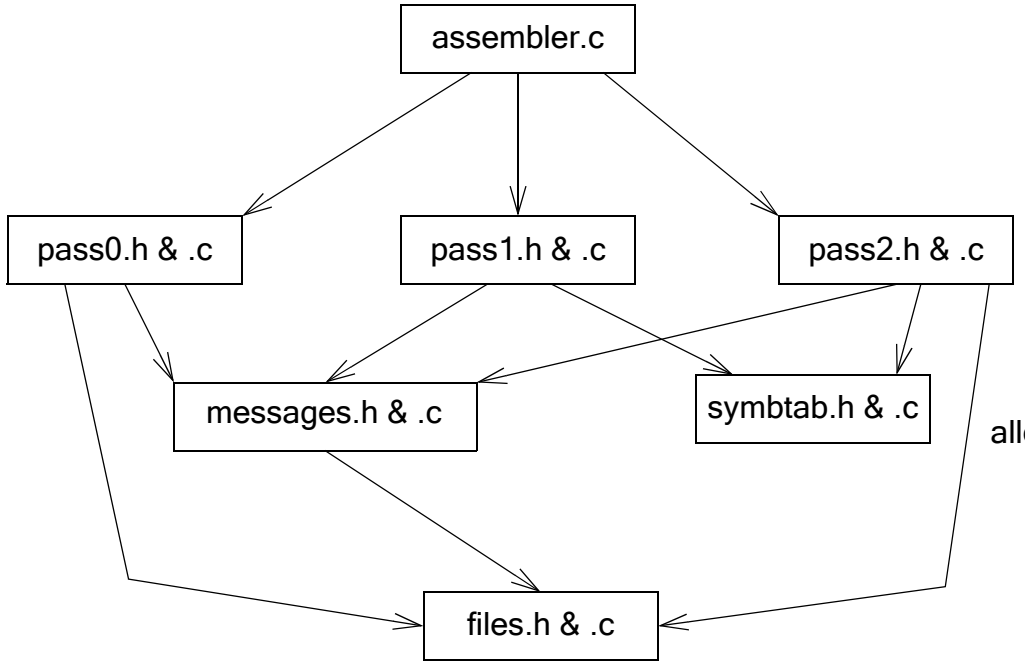
Übersetzungs- und Bindevorgänge - Beispiel 2:

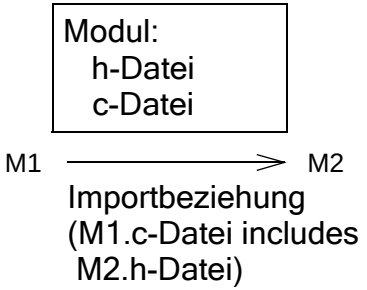


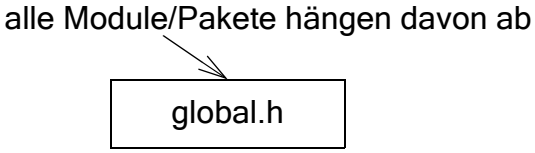


1. Schnittstelle (Typdefinition) von messages = messages.h ändert sich
2. [ggf. muss man auch messages.c ändern]
3. messages.c muss neu in messages.o übersetzt werden:
   cc -c messages.c
4. pass0.c, pass1.c und pass2.c müssen neu übersetzt werden
   cc -c pass0.c | cc -c pass1.c | cc -c pass2.c
5. Hauptprogramm assembler muss neu aus allen o-Dateien erzeugt werden:
   cc -o assemb assemb.o pass0.o pass1.o pass2.o messages.o files.o

Übersetzungs- und Bindevorgänge - Beispiel 3:







1. in global.h wird eine Konstante geändert
2. alle .c-Dateien müssen in beliebiger Reihenfolge (parallel) neu übersetzt werden:
   cc -c files.c | cc -c messages.c | cc -c symbtab.c | cc -c pass0.c | ...
3. Hauptprogramm assembler muss neu aus allen o-Dateien erzeugt werden:
   cc -o assemb assemb.o pass0.o pass1.o pass2.o messages.o files.o

Werkzeuge für das Build-Management (siehe auch [Po09], [Wie11]):

- Das Werkzeug „**make**“ (im folgenden im Detail präsentiert):
  - ⇨ deklarativer, regelbasierter Ansatz
  - ⇨ es werden Abhängigkeiten zwischen Entwicklungsartefakten definiert (mit zusätzlichen Kommandos zur Erzeugung abgeleiteter Artefakte)
  - ⇨ unabhängig v. Entwicklungsprozess, Programmiersprachen, Werkzeugen
- Das Werkzeug „**Ant**“ (make-Nachfolger im Java-Umfeld):
  - ⇨ gemischt deklarativer/imperativer Ansatz
  - ⇨ es werden Abhängigkeiten zwischen Aufgaben (Tasks) definiert
  - ⇨ Tasks können mit Kontrollstrukturen „ausprogrammiert“ werden
  - ⇨ verwendet XML-Syntax für Konfigurationsdateien
  - ⇨ in Java und vor allem für Java-Projekte entwickelt
  - ⇨ aber: unabhängig von Entwicklungsprozess, Projektstruktur, ...

Werkzeuge für das Build-Management - 2:

- Das Werkzeug „**Maven**“:
  - ⇨ deklarativer Ansatz mit relativ einfachen Konfigurationsdateien
  - ⇨ macht viele Annahmen über Entwicklungsprozess, Projektstruktur, ...
  - ⇨ keine frei definierbaren Regeln für bestimmte Sprachen, Werkzeuge sondern stattdessen Plugins
  - ⇨ Fokus liegt auf der Unterstützung von Java-Projekten
  - ⇨ holt sich Plugins, Libraries, ... aus zentralen Repositories
  - ⇨ oft kombiniert mit „Nexus“ für Einrichten lokaler (Cache-)Repositories
- Das Werkzeug „**Jenkins**“ (früher: „Hudson“):
  - ⇨ ergänzt Build-Werkzeuge wie Ant und Maven
  - ⇨ unterstützt die kontinuierliche Integration/Konstruktion von Produkten
  - ⇨ automatische Integration von Build-, Test-, Reporting-Werkzeugen

Das Programm „make“ zur Automatisierung von Build-Vorgängen:

- make wurde in den 70er Jahren „nebenbei“ von Stuart F. Feldman für die Erzeugung von Programmen unter Unix programmiert
- Varianten von make gibt es heute auf allen Betriebssystemplattformen (oft integriert in oder spezialisiert auf Compiler einer Programmiersprache)
- make werden durch ein sogenanntes makefile Erzeugungsabhängigkeiten zwischen Dateien mitgeteilt
- make benutzt **Zeitmarken** um festzustellen, ob eine Datei noch aktuell ist (Zeitmarke jünger als die Zeitmarken der Dateien, von denen sie abhängt)

Aufbau einer Abhängigkeitsbeschreibung:

```
ziel : objekt1 objekt 2 objekt3 ...
    Kommando zur Erzeugung von ziel aus objekt1 ...
  # Achtung Kommandozeile muss mit <tab> = Tabulator beginnen
```

Abhängigkeitsbeschreibungen für unser Beispiel:

```
assembler : assembler.o \
            pass0.o pass1.o pass2.o pass3.o \
            messages.o symbtab.o \
            files.o
    cc -o assembler   assembler.o pass0.o pass1.o pass2.o pass3.o \
                      messages.o symbtab.o files.o

assembler.o : assembler.c global.h pass0.h pass1.h pass2.h pass3.h
    cc -c assembler.c

pass0.o : pass0.h pass0.c global.h messages.h
    cc -c pass0.c

...
```

Achtung:

Das Zeichen „\“ muss immer letztes Zeichen einer fortgesetzten Zeile sein!

Aufruf von make ohne Parameter nach Änderung von messages.c:

1. der Aufruf sucht eine Datei namens makefile und soll eine aktuelle Version des Ziels der ersten Abhängigkeitsbeschreibung erzeugen
2. erstes Ziel ist assembler, also wird überprüft ob diese Datei noch aktuell ist
3. Dafür wird zunächst rekursiv überprüft, ob alle o-Dateien, von denen assembler abhängt, aktuell sind (wenn sie bereits existieren)
4. ...
5. messages.o hängt nur von Dateien ab, für die es keine eigenen Regeln gibt. Deshalb bricht der rekursive Prozess ab und es wird nur die Zeitmarke von messages.o mit den Zeitstempeln von messages.c, ... verglichen
6. messages.o hat eine ältere Zeitmarke als messages.c und wird deshalb durch Übersetzung neu erzeugt
7. assembler besitzt nun eine ältere Zeitmarke als messages.o und wird deshalb neu mit dem Binder (Linker) erzeugt

Makefiles mit mehreren Zielen:

```
assembler-unix :    assembler.o \
                    pass0.o pass1.o pass2.o pass3.o \
                    messages.o symbtab.o files-unix.o
    cc -o assembler assembler.o pass0.o pass1.o pass2.o pass3.o \
                    messages.o symbtab.o files-unix.o
    # make oder make assembler-unix führt den obigen Bindevorgang aus

assembler-windows : assembler.o \
                    pass0.o pass1.o pass2.o pass3.o \
                    messages.o symbtab.o files-windows.o
    cc -o assembler assembler.o pass0.o pass1.o pass2.o pass3.o \
                    messages.o symbtab.o files-windows.o
    # make assembler-windows führt den obigen Bindevorgang aus

cleanup :
    rm  assembler.o pass0.o pass1.o pass2.o pass3.o \
        messages.o symbtab.o files.o
    # make cleanup löscht immer o-Dateien, da Datei cleanup nicht existiert
```

Besseres Makefile mit selbstdefinierten Makros:

```
CC = cc                 # Aufruf des C-Compilers
CFLAGS = -c             # Flag für Übersetzung
LD = cc                 # Name des Binders (hier in C-Compiler integriert)
LDFLAGS = -o            # Flags für Binden
DEBUG =                 # Debugoption deaktiviert; Aktivierung als -g

assembler : assembler.o \
            pass0.o pass1.o pass2.o pass3.o \
            messages.o symbtab.o \
            files.o
    ${LD} ${DEBUG} ${LDFLAGS}  assembler assembler.o \
                               pass0.o pass1.o pass2.o pass3.o \
                               messages.o symbtab.o files.o

assembler.o : assembler.c global.h pass0.h pass1.h pass2.h pass3.h
    ${CC} ${DEBUG} ${CFLAGS} assembler.c
```

Weniger redundantes Makefile mit sogenannten internen Makros:

```
...
assembler : $@.o   pass0.o pass1.o pass2.o pass3.o \
                   messages.o symbtab.o files.o
        ${LD} ${DEBUG} ${LDFLAGS} $@ $^

assembler.o :$*.c  global.h pass0.h pass1.h pass2.h pass3.h
     ${CC} ${DEBUG} ${CFLAGS} $*.c
```

Erläuterung der seltsamen Sonderzeichen-Makros:

- $@ bezeichnet immer das Ziel einer Regel mit Suffix (z.B. .o oder .c)
- $* bezeichnet immer das Ziel einer Regel ohne Suffix (assembler.o wird also zu assembler verkürzt)
- $^ bezeichnet alle Objekte einer Regel rechts vom „:“ (z.B. assembler.o ...)
- $? bezeichnet alle Objekte einer Regel rechts vom „:“, die neuer als das Ziel sind
- $< bezeichnet das erste Objekt rechts vom „:“

Muster-Suche und Substitutionen in make:

Oft benötigt man auf der rechten Seite einer Regel oder in der Kommandozeile alle Dateinamen (im aktuellen Verzeichnis), die einem bestimmten **Namensmuster** entsprechen (also etwa mit einem bestimmten Suffix enden).

```
$(wildcard Muster)
```

liefert bzw. **sucht** alle Dateien im aktuellen Verzeichnis, die dem angegebenen Muster entsprechen. Das Muster kann beispielsweise folgende Form haben:

```
$(wildcard *.c)
```

liefert alle Dateien zurück, die das Suffix „.c“ besitzen (der „*“ matcht alles).

Will man in einer Liste von Dateinamen (durch Leerzeichen getrennte Strings) einen Teilstring durch einen anderen Teilstring **ersetzen** (substituieren) verwendet man:

```
$(patsubst Suchmuster, Ersatzstring, Liste von Wörtern)
```

Folgender Ausdruck ersetzt alle „.c“-Suffixe durch „.o“-Suffixe:

```
$(patsubst %.c, %.o, Liste von Wörtern)
```

Das Zeichen „%“ bezeichnet den gleichbleibenden Teil bei der Ersetzung.

Einsatz von Muster-Suche und -Substitution:

```
...

h-files = $(wildcard *.h)
# alle h-Dateien im aktuellen Verzeichnis werden der Var. h-files zugewiesen

o-files = $(patsubst %.c, %.o, $(wildcard *.c))
# alle c-Dateien im aktuellen Verzeichnis werden gefunden und mit
# ausgetauschtem Suffix (c wird gegen o getauscht) o-files zugewiesen

assembler : ${o-files}
        ${LD} ${DEBUG} ${LDFLAGS} $@ $^

assembler.o :$*.c  ${h-files}
     ${CC} ${DEBUG} ${CFLAGS} $*.c
```

Muster-Regeln in make:

Manchmal will man Regeln schreiben, die alle Dateien mit einem bestimmten Präfix oder Suffix aus einer anderen Datei mit unterschiedlichem =Präfix oder Suffix aber ansonsten gleichem Namen errechnen. So wird z.B. jede c-Datei in eine o-Datei ansonsten gleichen Namens übersetzt. Die Muster-Regel ist für die Definition solcher Abhängigkeiten geeignet:

```
prefix1%suffix1 : prefix1%.suffix2 ...
    Kommando
```

Beispiel für Muster-Regel:

Übersetzung von c-Dateien in o-Dateien geht wie folgt:

```
...

%.o : %.c ${h-files}
    ${CC} ${DEBUG} ${CFLAGS} $<
```

Ein letztes Beispiel für makefiles:

Es soll ein makefile geschrieben werden, das folgende Anforderungen erfüllt:

- beim Aufruf von make ohne Parameter sollen im aktuellen Verzeichnis alle Dateien mit der Endung gif in Dateien mit der Endung jpg übersetzt werden; dafür wird das Kommando convert verwendet
- das Konvertieren soll nur passieren, wenn die jpg-Datei zu einer gif-Datei noch nicht existiert oder einen älteren Zeitstempel besitzt
- alle jpg-Dateien mit zugehöriger gif-Datei werden zu einer Datei (einem Archiv) images.zip zusammengepackt mit dem Kommando zip
- die zip-Datei wird nur erzeugt, wenn vorher mindestens eine jpg-Datei neu erzeugt wurde (nur neue jpg-Dateien werden im Archiv ausgetauscht)
- für ein einfaches makefile können sie davon ausgehen, dass das aktuelle Verzeichnis nur die Dateien tobias.gif, johannes.gif, markus.gif und andy.jpg enthält
- ein „fortgeschrittenes“ makefile muss für beliebig viele gif- und jpg-Dateien funktionieren

Einfaches makefile für bekannte Dateien:

```
default : images.zip

images.zip : tobias.jpg johannes.jpg markus.jpg
  zip -u images.zip tobias.jpg johannes.jpg markus.jpg

tobias.jpg : tobias.gif
  convert tobias.gif tobias.jpg

johannes.jpg : johannes.gif
  convert johannes.gif johannes.jpg

markus.jpg : markus.gif
  convert markus.gif markus.jpg
```

Besseres makefile für beliebige Dateien:

```
jpg-files = $(patsubst %.gif, %.jpg, $(wildcard *.gif))

default : images.zip

images.zip : ${jpg-files}
  zip -u $@ $^

%.jpg : %.gif
  convert $? $@
```

Variante für inkrementelle Aktualisierung von images.zip:

Will man nur die veränderten jpg-Dateien neu in das Archiv aufnehmen, dann ist eine Regel wie folgt zu ändern:

```
images.zip : ${jpg-files}
  zip -u $@ $?
```

Bewertung des Buildmanagements mit make:

- nur ein Bruchteil der Funktionen von make wurde vorgestellt
- ursprüngliches make führt Erzeugungsprozesse sequentiell aus
  - ⇨ GNU make kann Arbeit auf mehrere Rechner verteilen und parallel durchführbare Erzeugungsschritte gleichzeitig starten
- Steuerung durch Zeitmarken ist sehr „grob“:
  - ⇨ **zu häufiges neu generieren**: irrelevante Änderungen (wie Ändern von Kommentaren für Übersetzung) werden nicht erkannt
  - ⇨ **zu seltenes neu generieren**: Änderung von Übersetzerversionen, Übersetzungsoptionen etc. werden nicht erkannt
- kaum Verzahnung mit Versionsverwaltung:
  - ⇨ make wird in aller Regel auf eigenem Repository durchgeführt
  - ⇨ erzeugte/abgeleitete Objekte werden also immer privat gehalten
- makefiles selbst müssen manuell aktuell gehalten werden
  - ⇨ programmiersprachenspezifisches makedepend erzeugt makefiles

Erzeugung von Makefiles:

1. in jedem Quelltextverzeichnis gibt es ein „normales“ Makefile, das den Übersetzungsprozess mit make in diesem Verzeichnis steuert
2. „normale“ Makefiles werden von makedepend durch Analyse von Quelltextdateien erzeugt (in C werden includes von .h-Dateien gesucht)
3. ein „Super“-Makefile sorgt dafür, dass
   - ⇨ makedepend nach relevanten Quelltextänderungen in den entsprechenden Verzeichnissen aufgerufen wird
   - ⇨ in allen „normalen“ Makefiles dieselben Makrodefinitionen verwendet werden (soweit gewünscht)
   - ⇨ geänderte Makefiles oder Makrodefinitionen dazu führen, dass in den betroffenen Verzeichnissen alle abgeleiteten Objekte neu erzeugt werden

Achtung:

- ☞ Makefile-Erzeugung ist eine „Wissenschaft“ für sich, wenn viele verschiedenen Übersetzer und Generatoren in einem Projekt verwendet werden

Generierung von Makefiles mit GNU Autotools [Ca10]:

Die GNU **Autotools** sind mehrere Werkzeuge, die aufbauend auf make das Build-Management, also die Erstellung und Installation von Softwarekonfigurationen für verschiedenste Zielplattformen, erleichtern.

- Fokus liegt auf Softwareprojekten, in denen vor allem C, C++ oder Fortran (77) als Programmiersprachen eingesetzt werden
- die Werkzeuge eignen sich nicht (kaum) für Java-Projekte oder SW-Entwicklung im Windows-Umfeld
- **Automake** unterstützt Generierung „guter“ makefiles aus deutlich kompakteren Konfigurationsdateien (die von allen populären make-Varianten ausführbar sind)
- **Autoconf** unterstützt Konfigurationsprozess von Software durch Generierung von config.h-Skelette, Konfigurationsskripte, ...
- schließlich gibt es noch **Libtool**, das Umgang mit Bibliotheken erleichtert

Zusammenfassung und Ratschläge:

- Bereits in kleinsten Projekten ist die Automatisierung von Erzeugungsprozessen (Buildmanagement) ein Muss; in einfachen Fällen bietet der verwendete Compiler die notwendige Unterstützung.
- Im Linux/Unix-Umfeld ist „(GNU) make“ das Standardwerkzeug für die Automatisierung von Erzeugungsprozessen.
- In jedem Projekt sollte es genau einen Verantwortlichen für die Pflege von Konfigurationsdateien (Makefiles) und Build-Prozesse geben.
- Für Java-Programmentwicklung gibt es mit Ant ein speziell zugeschnittenes moderneres „Build-Tool“; siehe http://jakarta.apache.org/ant/index.html.
- Noch „moderner“ sind Maven und Jenkins für „Continuous Integration“, also die automatisierte, permanente Erzeugung von Software-Releases.
- Nicht generierte (Anteile von) Konfigurationsdateien müssen selbst unbedingt der Versionsverwaltung unterworfen werden.

2.6 Änderungsmanagement

Ein festgelegter Änderungsmanagementprozess sorgt dafür, dass Wünsche für Änderungen an einem Softwaresystem protokolliert, priorisiert und kosteneffektiv realisiert werden.

Änderungsmanagement frei nach [So01]:

```
Ausfüllen eines Änderungsantragsformulars;
Analyse des Änderungsantrags;
if Änderung notwendig (und noch nicht beantragt) then
    Bewertung wie Änderung zu implementieren ist;
    Einschätzung der Änderungskosten;
    Einreichen der Änderung bei Kommission;
    if Änderung akzeptiert then
        Durchführen der Änderung für Release ...
    else
        Änderungsantrag ablehnen
else
    Änderungsantrag ablehnen
```

Änderungsmanagement frei nach [Wh00]:

1. ein Änderungswunsch (feature request) oder eine Fehlermeldung (bug report) wird eingereicht (Status **submitted**)
2. ein eingereichter Änderungswunsch wird evaluiert und dabei
   - ⇨ entweder abgelehnt (Status **rejected**)
   - ⇨ oder als Duplikat erkannt (Status **duplicate**)
   - ⇨ oder mit Kategorie und Priorität versehen (Status **accepted**)
3. ein akzeptierter Änderungswunsch wird von dem für seine Kategorie Zuständigen
   - ⇨ für ein bestimmtes Release zur Bearbeitung freigegeben (Status **assigned**)
   - ⇨ oder vorerst aufgeschoben (Status **postponed**)
4. ein zugewiesener Änderungswunsch wird von dem zuständigen Bearbeiter in Angriff genommen (Status **opened**)
5. irgendwann ist die Bearbeitung eines Änderungswunsches beendet und die Änderung wird zur Prüfung freigegeben (Status **released**)
6. erfüllt die durchgeführte Änderung den Änderungswunsch, so wird schließlich der Änderungswunsch geschlossen (Status **closed**)

Funktionalität von Änderungsmanagement-Werkzeugen:

- Änderungswünsche können über Browserschnittstelle übermittelt werden
- Änderungswünsche werden in Datenbank verwaltet
- Betroffene werden vom Statuswechsel eines Änderungswunsches benachrichtigt
- Integration mit Projektmanagement und KM-Management
- Trendanalyse und Statistiken als Grafiken (Anzahl neuer Fehlermeldungen, ...)

Werkzeuge für das Änderungsmanagement:

- „Open Software“-Produkt **Sourceforge** (GForge), das cvs mit Änderungs-managementdiensten kombiniert, siehe http://sourceforge.net
- „Open Software“-Produkt **Bugzilla** für reines Änderungsmanagement, siehe http://bugzilla.mozilla.org/
- kommerzielles Produkt **ClearQuest** von Rational/IBM (passend zu KM-Werkzeug ClearCase; siehe Abschnitt 2.7)
- flexibles Projektmanagement-Werkzeug **Redmine** (http://www.redmine.org/)

2.7 Verteilte Softwareentwicklung

Die bisher vorgestellten KM-Systeme (Versionsverwaltungssysteme) eignen sich nur bedingt für das Management sehr großer und geographisch verteilter Projekte.

Ein kommerzielles Produkt wie ClearCase (Rational/IBM) bietet für die verteilte Softwareentwicklung in großen Projekten u.a. folgende Vorteile gegenüber cvs und Subversion (siehe auch [Wh00]):

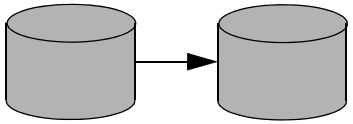
- Repositories (Version Object Bases = VOBs) können über beliebig viele Fileserver transparent verteilt werden
- Replica von Dokumentversionen können auf beliebig vielen Fileservern gehalten werden (ein Server hält aber das Original = Master mit Schreibrecht)
- Entwickler können entweder
  - ⇨ im **isolierten Workspace** mit eigenen Objekten arbeiten (hoher Speicherplatzverbrauch, geringere Netzlast) oder
  - ⇨ über **dynamische Sichten** auf gemeinsamen Repository arbeiten (geringerer Speicherplatzverbrauch, höhere Netzlast)

Weitere für uns neue Funktionen von ClearCase:

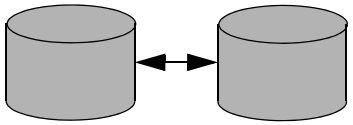
- Versionierung kann sogar transparent hinter Filesystem-Schnittstelle erfolgen
- nicht nur Quelltexte werden versioniert, sondern auch daraus abgeleitete Objekte mit allen zugehörigen Erzeugungsinformationen
- häufig benötigte abgeleitete Objekte werden im „Cache“ gehalten, weniger häufig benötigte bei Bedarf rekonstruiert (neu erzeugt)
- auch Verzeichnisse (Directories) unterliegen der Versionierung
- es können neue Attribute für bestimmte Arten versionierter Objekte eingeführt werden
- propiertäre Diff- und Merge-Algorithmen einzelner Werkzeuge können integriert werden
- ...

Gebräuchliche Verteilungsarchitekturen für ClearCase-Repositories:

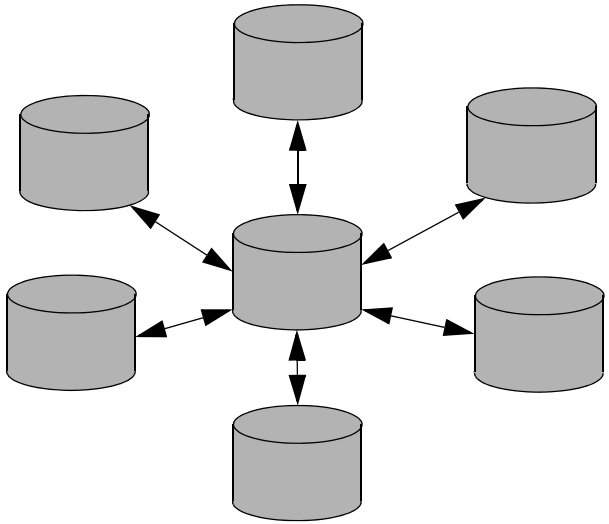
One Way Propagation:



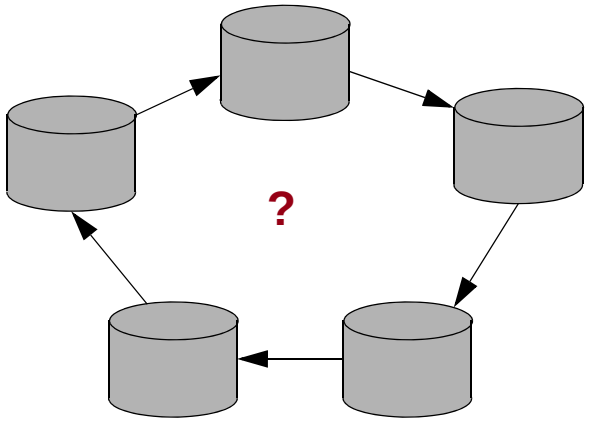
Peer to Peer Solution:



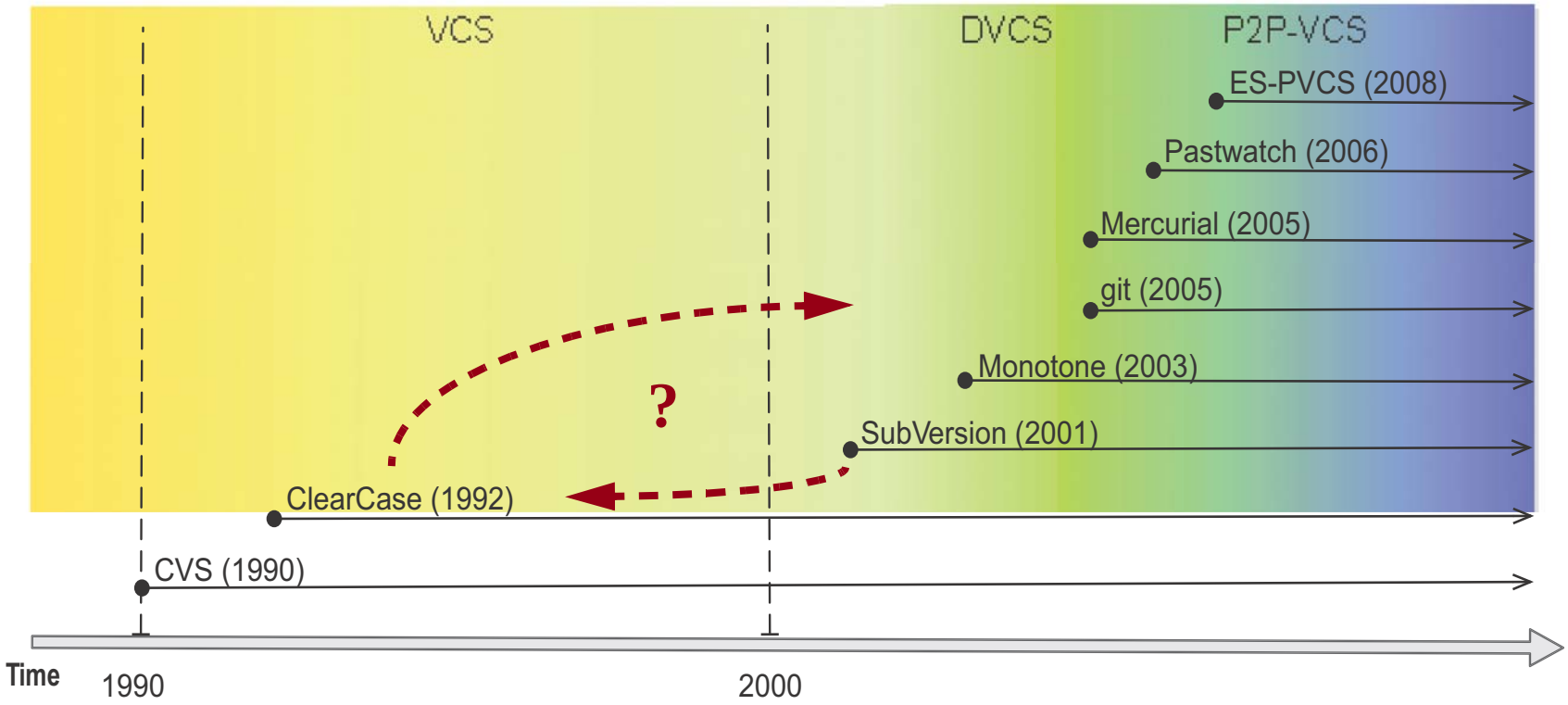
Centralized Main Repository (Star):



Circular Propagation:



Geschichte der (verteilten) Versionierungs-Systeme:



- **VCS** = **V**ersion **C**ontrol **S**ystem
- **DVCS** = **D**istributed **V**ersion **C**ontrol **S**ystem

WWW-Seiten einiger DVCS-Systeme (in Entwicklung):

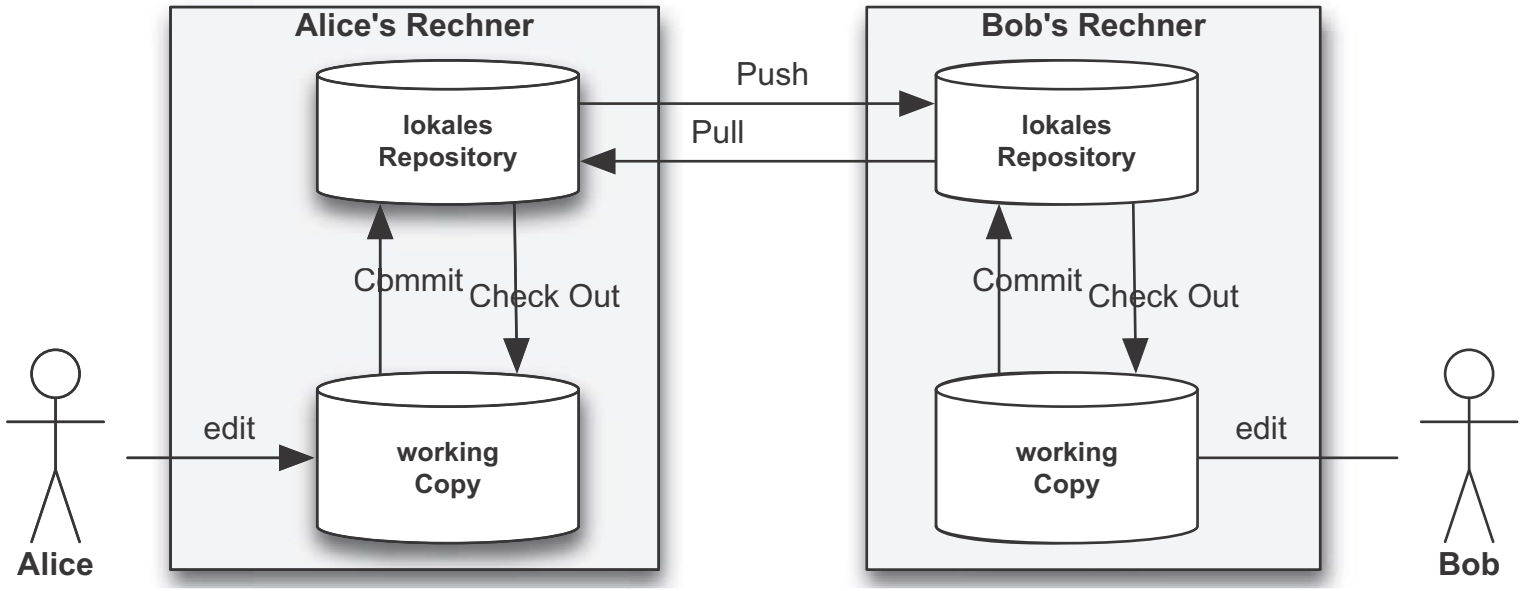
- Monotone: http://monotone.ca/
- Mercurial (aka hg): http://www.selenic.com/mercurial/wiki/
- Bazaar: http://bazaar-vcs.org/
- git: http://git-scm.com/

Anmerkung zu ClearCase:

Das System ist ein Grenzfall zwischen VCS und DVCS. Es handelt sich zwar um kein reines Client-Server-System mehr, aber die Verteilung auf mehrere Server und deren Kooperation hat doch noch sehr zentralistischen Charakter im Gegensatz zu den „reinen“ DVCS, die auf dem Rechner eines jeden Entwicklers eine eigene Instanz laufen haben.

„Echt“ verteilte Versionierungs-Systeme:

Jeder Benutzer hat sein eigenes vollständiges Repository (statt globale Repositories für Teilgruppen von Entwicklern wie bei ClearCase).



- **Push**: Revisionen zu anderen Repositories aktiv propagieren
- **Pull**: Revisionen von anderen Repositories aktiv holen

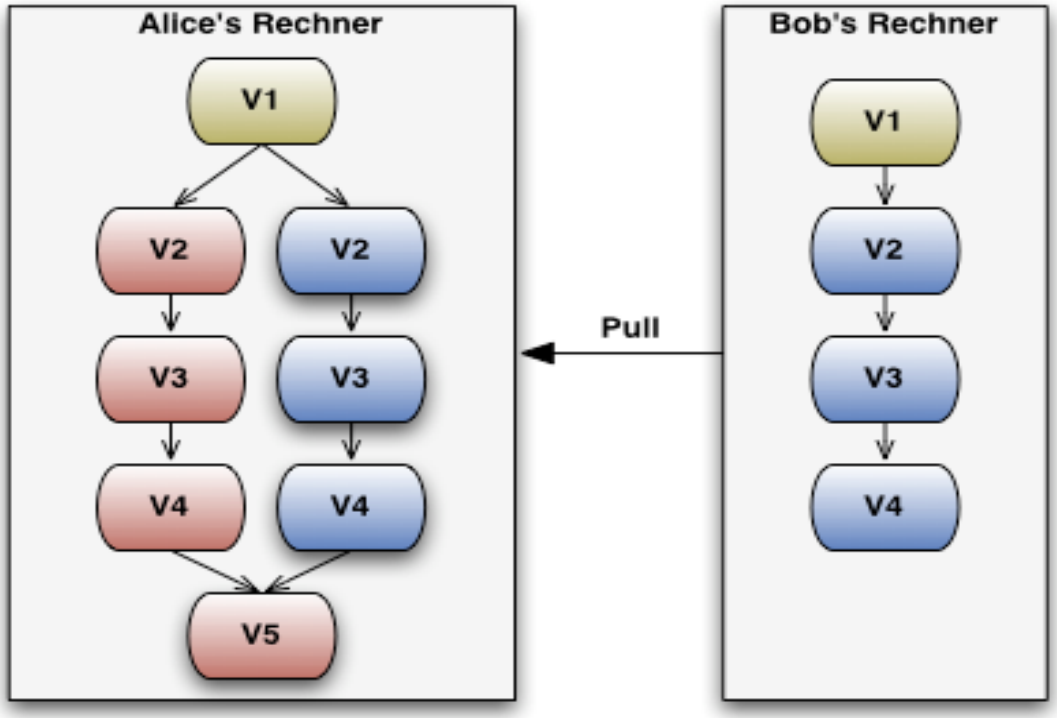
Prinzipien „echt“ verteilter Versionierungs-Systeme wie „git“:

- Jeder Entwickler hat eigenes lokales Versionierungs-Repository mit Snapshots wie bei SVN (Subversion) zusätzlich zu üblichen Arbeitskopien von Dateien
- Check-Out, Commit, ... arbeiten erst mal nur auf eigenen lokalem Repository
- Austausch zwischen Repositories erfolgt mit Push und Pull (zwischen bekannten Personen/Rechnern) via Email oder ssh oder ...
- Bei Push/Pull werden parallele Revisionen angelegt, die dann mit „merge“ zusammengeführt werden (müssen)

Bewertung:

- ☺ Entwickler können auch „offline“ mit eigenem Repository arbeiten
- ☺ Änderungen können zu selbst gewähltem Zeitpunkt integriert werden (merge)
- ☹ Nach Platten-Crash ist lokales Repository weg
- ☹ Hoher Aufwand für Verteilung von Änderungen an andere Entwickler (merge)

Ablauf der Propagation von Änderungen:



- Änderungen (Revisionen) von Bob werden als paralleler Zweig v. Alice „gepullt“
- Alice „merged“ dann ihre neueste Revision V4 mit neuester Revision V4 von Bob

Verteilte „Peer-to-Peer“-Versionierung-Systeme:

- **Virtuell** existiert ein **gemeinsames globales Repository** für alle Entwickler
- Es gibt dennoch keinen ausgewählten zentralen Server
- Tatsächlich besitzt jeder Peer ein eigenes Repository (mit allen Dateien oder ausgewählter Menge von Revisionen)
- Das P2P-Versionierungs-System ist „immun“ gegen Ausscheiden einzelner Peers (Revisionen werden auf mehreren Peers gehalten)
- Bei **Netzpartitionierungen** (Spezialfall: einzelner Entwickler arbeitet „offline“) wird je Partition ein eigener Branch angelegt
- Schließen sich Partitionen wieder zusammen, dann werden Branches wieder verschmolzen (im allgemeinen mit Entwicklerhilfe)

Fazit:

P2P-Versionierungssysteme sollen die Vorteile von VCS und DVCS vereinen!

2.8 Zusammenfassung

Mit einem für die jeweilige Projektgröße sinnvollen KM-Management steht und fällt die Qualität eines Softwareentwicklungsprozesses, insbesondere nach der Fertigstellung des ersten Softwarereleases.

Meine Ratschläge für das KM-Management:

- besteht ihr System aus mehr als einer Handvoll Dateien, so sollte ein **Buildmanagementsystem** wie make/Ant/Maven verwendet werden
- haben sie Entwicklungszeit von mehr als ein paar Tagen oder mehr als einem Entwickler, so ist ein **Versionsmanagementsystem** wie svn oder git ein Muss
- haben sie mehr als einen Anwender oder mehr als ein Release der Software, so ist ein **Changemanagementsystem** wie in Bugzilla/Redmine einzusetzen
- svn/git, make/... , Bugzilla sind das Minimum für jedes „ernsthafte“ Projekt, Sourceforge mit Kombination von svn/git und Bugzilla-Funktionalität sinnvoll
- ... alles weitere hängt von der Projektgröße und -art ab

2.9 Zusätzliche Literatur

[Ca10] J. Calcote: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool; William Pollock Publ. (2010)

[Wh00] B. Collins-Sussman, B.W. Fitzpatrick, C.M. Pilato: Version Control with Subversion, Version 1.1 (2004); http://svnbook.red-bean.com

[CW98] R. Conradi, B. Westfechtel: *Version Models for Software Configuration Management*, ACM Computing Surveys, vol. 30, no. 2, ACM Press (1998), 232-282

[DIN96] DIN EN ISO 10007:1996 Leitfaden für Konfigurationsmanagement

[IEEE88] IEEE Standard 1042-1987: *IEEE Guide to Software Configuration Management*, IEEE Computer Society Press (1988)

[IEEE98] IEEE Standard 828-1998: *IEEE Standard for Software Configuration Management Plans*, IEEE Computer Society Press (1998)

[Po09] G. Popp: *Konfigurationsmangement mit Subversion, Maven und Redmine*, dpunkt.verlag, 3. Auflage (2009)

[PBL05] K. Pohl, G. Böckle, F. van der Linden: *Software Product Line Engineering - Foundations, Principles, and Techniques*, Springer Verlag 2005, 467 Seiten

[Ve00] G. Versteegen: *Projektmanagement mit dem Rational Unified Process*, Springer-Verlag (2000)

[Ve03] G. Veersteegen (Hrsg.), G. Weischedel: *Konfigurationsmanagement*, Springer-Verlag (2003)

[VSH01] G. Versteegen, K. Salomon, R. Heinhold: *Change Management bei Software-Projekten*, Springer-Verlag (2001)

[Wh00] A. Zeller, J. Krinke: *Programmierwerkzeuge: Versionskontrolle - Konstruktion - Testen - Fehlersuche unter Linux*, dpunkt-Verlag (2000)

[Wie11] S. Wiest: *Continuous Integration mit Hudson/Jenkins: Grundlagen und Praxiswissen für Einsteiger und Umsteiger*, dpunkt.verlag (2011)

3. Statische Programmanalyse & Metriken

Statische Codeanalyse verlangt ein stures, monotones Anwenden von relativ einfachen Regeln auf oft umfangreichen Code. Diese Aufgabe erfordert keinerlei Kreativität aber eine sehr große Übersicht und Kontrolle. ... Statische Codeanalyse ist daher prädestiniert zur Automatisierung durch Werkzeuge. Ich empfehle Ihnen, diese Techniken entweder werkzeuggestützt oder gar nicht einzusetzen. [Li02]

Lernziele:

- verschiedene Arten der statischen Programmanalyse kennenlernen
- einige Werkzeuge zur statischen Programmanalyse einsetzen können
- Zusammenhänge zwischen Softwarequalität und Analyseverfahren verstehen
- strukturorientierte Analyse- und Messverfahren lernen

3.1 Einleitung

- Oft (fast immer) finden sich **80% aller Probleme** mit einem Softwaresystem in 20% des entwickelten Codes.
- Die statische Programmanalyse versucht meist werkzeuggestützt frühzeitig die **problematischen 20%** eines Softwaresystems zu finden.
- Statische Analyseverfahren identifizieren Programmteile von **fragwürdiger Qualität** und liefern damit Hinweise auf potentielle Fehlerstellen.
- Statische Analyseverfahren **versuchen** die Qualität von Software zu **messen**, können deshalb zur Festlegung von Qualitätsmaßstäben eingesetzt werden.
- Die statische Programmanalyse setzt **keine vollständig ausführbaren** Programme voraus.
- Die statische Programmanalyse kann also **frühzeitig** bei der Neuentwicklung und kontinuierlich bei der Wartung eines Softwaresystems eingesetzt werden.

Arten von Analyseverfahren - zur Erinnerung:

- **analysierende Verfahren (statische Programmanalyse)**:

  der „Prüfling” (Programm, Modell, Dokumentation) wird von Menschen oder Werkzeugen auf Vorhandensein/Abwesenheit von Eigenschaften untersucht

  - ⇨ **Inspektion** (Review, Walkthrough): organisiertes „Durchlesen” in Gruppe
  - ⇨ **statische Analyse**: werkzeuggestützte Ermittlung von „Anomalien”
  - ⇨ **Verifikation**: werkzeuggestützter Beweis von Eigenschaften

- **testende Verfahren (dynamische Programmanalyse)**:

  der „Prüfling” wird mit konkreten oder abstrakten Eingabewerten von Menschen oder Werkzeugen ausgeführt

  - ⇨ **dynamischer Test** (Simulation): Ausführung mit konkreten Eingaben
  - ⇨ **symbolischer Test**: Interpretation mit symbolischen Eingaben (die oft unendliche Mengen möglicher konkreter Eingaben repräsentieren)

Arten der statischen Programmanalyse - 1:

- **Visualisierung von Programmstrukturen**: unästhetisches Layout liefert Hinweise auf sanierungsbedürftige Teilsysteme
  - ⇨ siehe Abschnitt 3.2 über Softwarearchitekturen und Visualisierung
- **manuelle Inspektionsverfahren**: organisiertes Durchlesen u. Diskutieren von Entwicklungsdokumenten (auch **Audit** genannt, ggf. mit Werkzeugunterstützung)
  - ⇨ siehe Abschnitt 3.3 über manuelle Inspektionsverfahren
- **Compilerprüfungen**: Syntaxprüfungen, Typprüfungen, ...
  - ⇨ sollten hinreichend bekannt sein
- **Programmverifikation und symbolische Ausführung**: Beweis der Korrektheit eines Programms mit Logikkalkül oder anderen mathematischen Mitteln
  - ⇨ siehe etwa Vorlesungen zu „Verifikationstechnologien“ (Eveking, Mantel)
- **Stilanalysen**: Programmierkonventionen für Programmiersprachen
  - ⇨ Java-Programmierkonventionen im „Software-Praktikum“ und ...

Beispiele stilistischer Regeln für C++ - 1:

- vermeide komplexe Zuweisungen:

```
// anstelle von i *= j++ + 20;
h = j + 20;
j++;        // j = j+1;
i *= h;     // i = i * h;
```

- nicht zu viele „Laufvariablen“ in Schleifen:

```
for ( i = 0, j = 0, k = 10, l = -1 ; i < cnt;   i++, j++, k--, l += 2  ) {
  // do something
}
```

besser:

```
l = -1;
for ( i = 0, j=0, k=10; i < cnt; i++, j++, k-- ) {
  // do something
  l += 2;
}
```

Beispiele stilistischer Regeln für C++ - 2:

- kein teilweises Überladen von Methoden aus Oberklasse:

```
class Animal {
public:
   void oper (int i);
   void oper (bool b);
};

class Elephant : Animal {
public:
   void oper (char c);
// fehlende Methode:  void oper (int i);
   void oper (bool b);
};
```

Lösung:

Methode „oper (int i)“ in Unterklasse Elephant mit aufführen; dann ist klar, dass wirklich oper für neuen Parametertyp eingeführt werden soll.

Beispiele stilistischer Regeln für C++ - 3:

- nie Attribute in Unterklasse mit selbem Namen wie in Oberklasse

```
class Animal {
  public int attr1;
};

class Elephant : Animal {
  public int attr1;
  // something...;
};
```

Grund:

Es werden zwei Attribute mit dem Namen attr1 deklariert; für Objekte der Klasse Elephant ist aber nur das neue Attribut attr1 direkt zugreifbar.

Lösung:

Anderen Namen verwenden ...

Beispiele stilistischer Regeln für C++ - 4:

- Beachte übliche Namenskonventionen:
  - ⇨ Klassennamen beginnen immer mit Großbuchstaben
  - ⇨ alle anderen Namen mit Kleinbuchstaben
  - ⇨ Variablennamen enthalten nur Kleinbuchstaben und Ziffern
- keine komplexen Ausdrücke in Schleifenbedingungen:

```
for (int i = 0; i < vector.size(); i++) {
   // do something
}
```

Besser:

```
vectorsize = vector.size();
for (int i = 0; i < vectorsize; i++) {
   // do something
}
```

- ...

Arten der statischen Programmanalyse - 2:

- **Kontroll- und Datenflussanalysen**: die Programmstruktur wird untersucht, um potentielle Zugriffe auf undefinierte Variablen, möglicherweise nie ausgeführten Code, etc. zu entdecken.

  ⇨ siehe Abschnitt 3.4

- **Metriken**: Programmeigenschaften werden gemessen und als Zahl repräsentiert - in der Hoffnung, dass kausaler Zusammenhang zwischen Softwarequalität (z.B. Fehlerzahl) und berechneter Maßzahl besteht.

  ⇨ siehe Abschnitt 3.5

Anmerkung:

In Abschnitt 3.5 mehr zur Bewertung von Metriken und Validierung von Hypothesen über Zusammenhang von Softwarequalität und Maßzahlen.

3.2 Softwarearchitekturen und -visualisierung

Große Systeme sind immer in Subsysteme gegliedert, von denen jedes eine Anzahl von Diensten bereitstellt. Der fundamentale Prozess zur Definition dieser Subsysteme und zur Errichtung eines Rahmenwerkes für die Steuerung und Kommunikation dieser Subsysteme wird **Entwurf der Architektur** ... genannt. [So01]

Begriffe nach Sommerville:

- Ein **Softwaresystem** besteht aus Teilsystemen, die zusammengehörige Gruppen von Diensten anbieten und möglichst unabhängig voneinander realisiert sind.
- Ein **Teilsystem** kann wiederum aus Teilsystemen aufgebaut werden, die aus Moduln (Paketen) bestehen.
- Ein **Modul** (Paket) bietet über seine Schnittstelle Dienste an und benutzt (importiert) zu ihrer Realisierung Dienste anderer Module (Pakete).
- Ein Modul fasst „verwandte“ Prozeduren, **Klassen**, ... zusammen.

Programmarchitekturen in C++:

- C++ besitzt keine Sprachmittel zur Deklaration von Teilsystemen. Oft werden **Dateibäume** (Subdirectories) zur Teilsystembildung eingesetzt.
- Module werden in C++ durch cpp/cc-**Dateien** realisiert;
  h(eader)-Dateien definieren wie in C die Schnittstellen von Modulen.
- **include**-Beziehungen (textuelles Einkopieren) von h-Dateien realisieren Modulimporte.

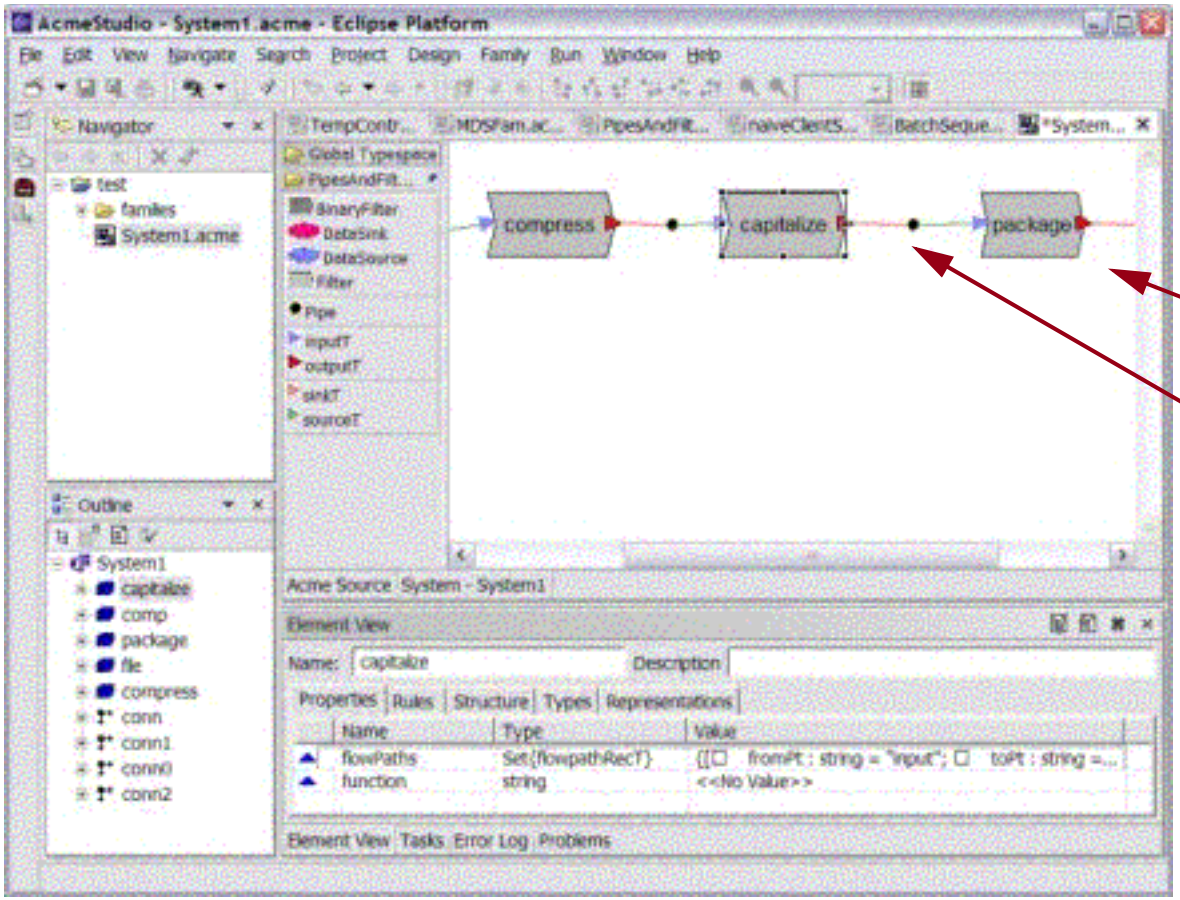
Zusätzliche Programmstrukturen in C++:

- Vererbungshierarchien auf Klassen
- sonstige Beziehungen zwischen Klassen
- Kontrollfluss eines Programms
- ...

Softwarearchitekturen sind mehr als Teilsysteme und Module:

- In den 80er Jahren wurden Programmarchitekturen mit sogenannten „**M**odule **I**nterconnection **L**anguages (**MIL**)“ definiert, die nur Module und Import-Beziehungen kennen.
- Seit den 90er Jahren werden auch „**A**rchitecture **D**escription **L**anguages“ (**ADL**s) eingesetzt, die Komponenten mit Eingängen und Ausgängen und Verbindungen dazwischen verwenden (siehe Vorlesung „Echtzeitsysteme“); siehe auch [SG96]
- Heute verwendet man einen noch umfassenderen Architekturbegriff; in „Software Engineering I“ werden folgende **Architektursichten** im Zusammenhang mit der „**U**nified **M**odeling **L**anguage“ (**UML**) eingeführt:
  - ⇨ Teilsystem-Sicht (Paketdiagramme)
  - ⇨ Struktur-Sicht (Klassendiagramme, Kollaborationsdiagramme)
  - ⇨ Kontrollfluss-Sicht (Aktivitätsdiagramme, ...)
  - ⇨ Datenfluss-Sicht (Aktivitätsdiagramme, ...)
  - ⇨ ...

Beispiel für ADL - Filterketten in ACME:

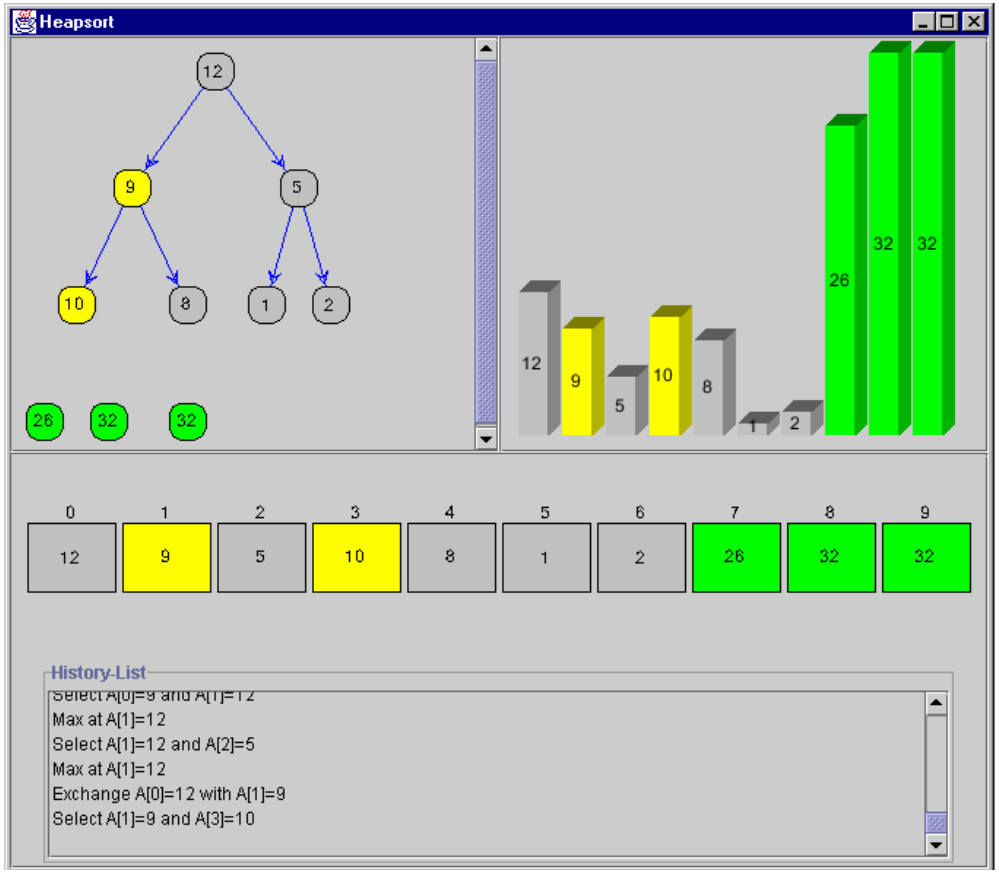


Acme-Studio-Werkzeug: http://www-2.cs.cmu.edu/~acme/AcmeStudio

Software-Visualisierung - Begriffsklärung - siehe auch [SD98]:

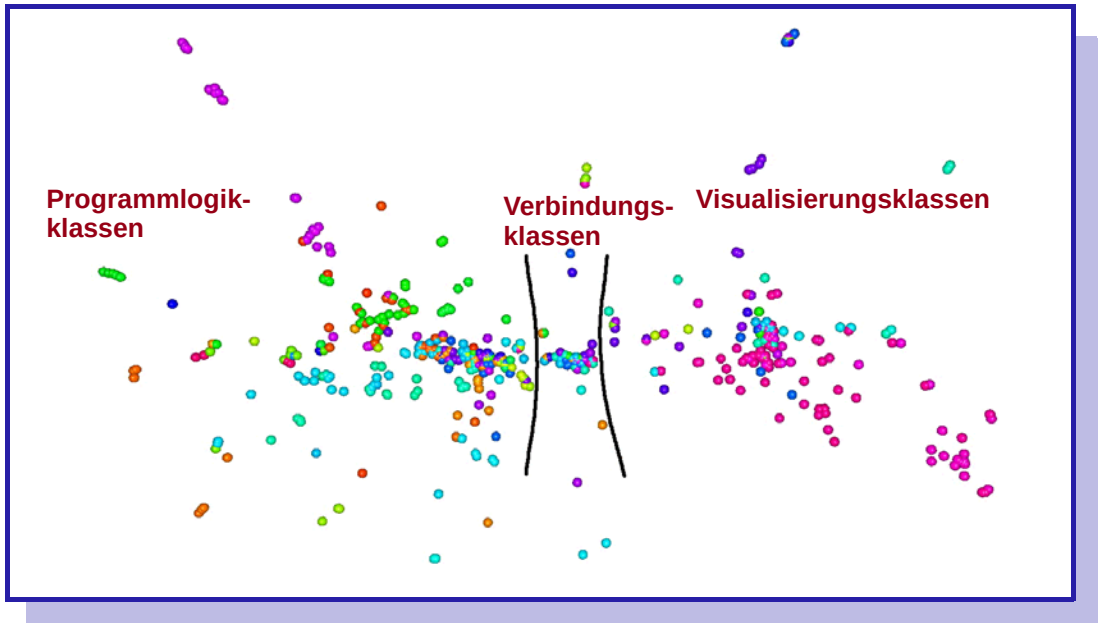
- **Algorithmus versus Programm**:
  - ⇨ Algorithmenvisualisierung: die prinzipielle Idee eines Algorithmus wird durch Visualisierung erläutert
  - ⇨ Programmvisualisierung: das Programm selbst (Implementierung eines Algorithmus) wird visualisiert
- **Struktur versus Verhalten**:
  - ⇨ Struktur/Aufbau von Algorithmus oder Programm wird visualisiert
  - ⇨ Verhalten: Ablauf eines Algorithmus oder Programms wird visualisiert
- **statisch versus dynamisch**:
  - ⇨ statisch: ein Bild/Diagramm wird zur Visualisierung verwendet
  - ⇨ dynamisch: Bildsequenz/Film wird zur Visualisierung eingesetzt
- **Programmvisualisierung versus visuelles Programmieren (Modellieren)**:
  - ⇨ Programmvisualisierung: Programm wird grafisch dargestellt
  - ⇨ visuelles Programmieren: es wird grafisch programmiert (modelliert)

Dynamische Algorithmenvisualisierung - Beispiel:



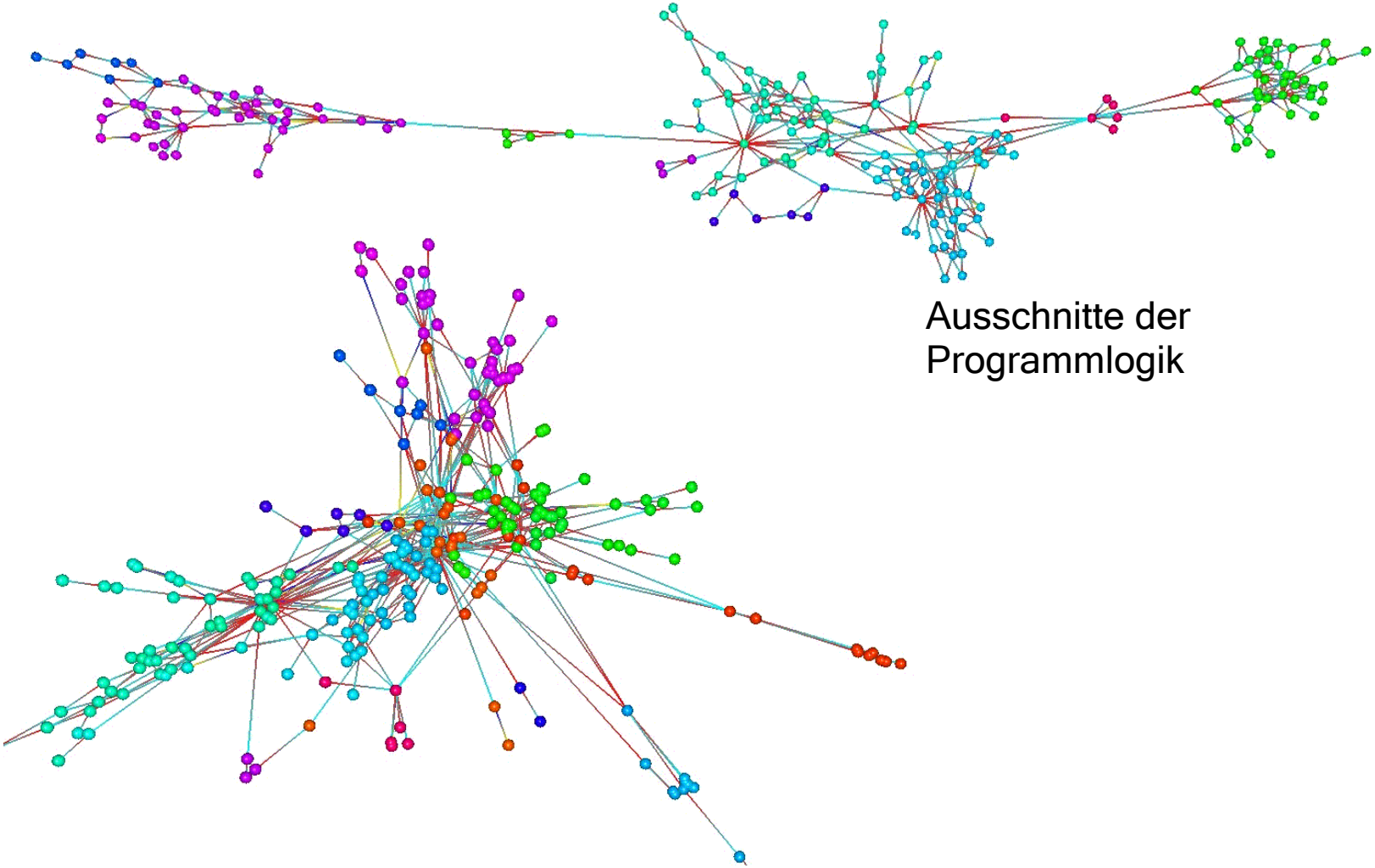
Dargestellt wird hier die Arbeitsweise des Sortieralgorithmus Heapsort (nicht mehr verfügbare Implementierung der TU Wien)

Statische Visualisierung von Programmstruktur mit CrocoCosmos :



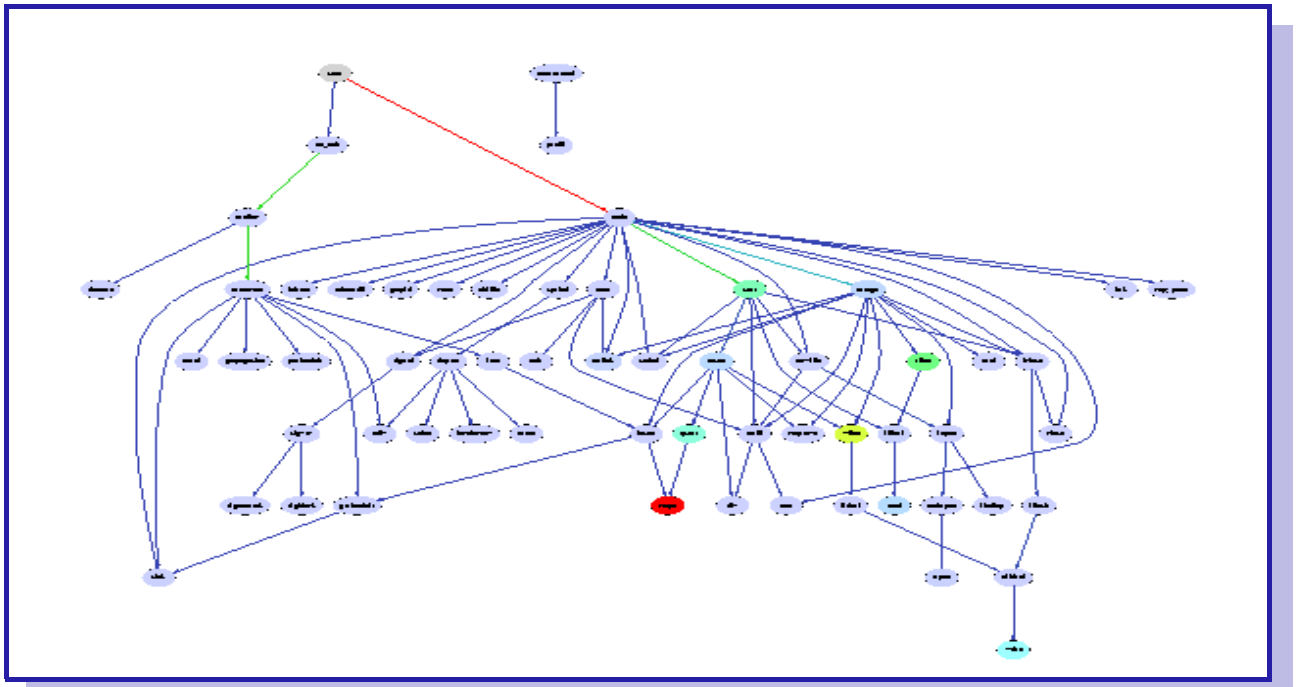
- grafische Darstellung eines Rahmenwerks für interaktive Anwendungen
- Klassen eines Teilsystems besitzen dieselbe Farbe, Beziehungen zwischen Klassen (aus Gründen der Übersichtlichkeit weggelasssen)
- siehe http://www-sst.informatik.tu-cottbus.de/CrocoCosmos/

Statische Visualisierung von Programmstrukturen - Fortsetzung:



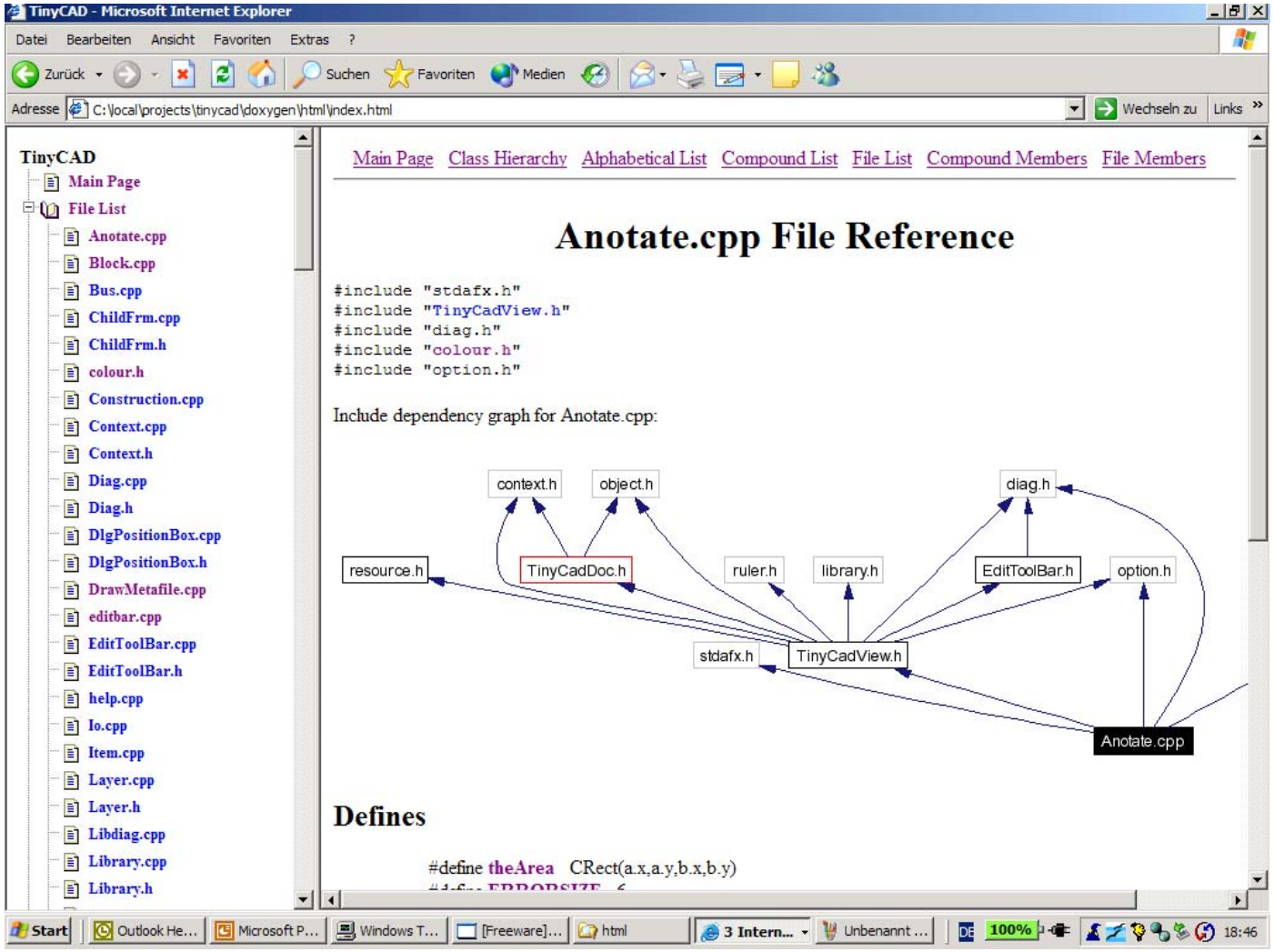
Ausschnitte der Programmlogik

Programmvisualisierung mit Doxygen und Graphviz:



- Doxygen generiert Programmdokumentation, Architekturdiagramme, ... ; siehe http://www.stack.nl/%7Edimitri/doxygen/index.html
- Graphviz (dot) ist Hilfsprogramm für Layoutberechnung; siehe http://www.graphviz.org/

Programmvisualisierung mit Doxygen - „include“-Abhängigkeiten:



Programmvisualisierung mit Doxygen - generierte Dokumentation:

3.3 Manuelle statische Testverfahren

Systematische Verfahren zur gemeinsamen „Durchsicht“ von Dokumenten (wie z.B. erstellte UML-Modelle, implementierten Klassen, …):

- **Programminspektion**: stark formalisiertes Verfahren bei dem Dokument nach genau festgelegter Vorgehensweise durch Gutachterteam untersucht wird; Fokus auf Fehlersuche (und nicht auf Diskussion alternativer Realisierungsmöglichkeiten)
- **(Technisches) Review**: weniger formale manuelle Prüfmethode; weniger Aufwand als bei Programminspektion, ähnlicher Nutzen; ggf. zusätzlicher Schwerpunkt auf Diskussion und Bewertung von Alternativen
- **Walkthrough (Informelles Review)**: unstrukturierte Vorgehensweise; Autor des Dokuments liest vor, Gutachter stellen spontan Fragen

Empirische Ergebnisse zu Programminspektion:

- Prüfaufwand liegt bei ca. 15 bis 20% des Erstellungsaufwandes
- 60 bis 70% der Fehler in einem Dokument können gefunden werden
- Nettonutzen: 20% Ersparnis bei Entwicklung, 30% bei Wartung

Psychologische Probleme bei analytischen Testverfahren:

- ☹ Entwickler sind in aller Regel von der Korrektheit der erzeugten Komponenten überzeugt (ihre Komponenten werden höchstens falsch benutzt)
- ☹ Komponententest wird als lästige Pflicht aufgefasst, die
  - ⇨ Folgearbeiten mit sich bringt (Fehlerbeseitigung)
  - ⇨ Glauben in die eigene Unfehlbarkeit erschüttert
- ☹ Entwickler will eigene Fehler (unbewusst) nicht finden und kann sie auch oft nicht finden (da ggf. sein Testcode von denselben falschen Annahmen ausgeht)
- ☹ Fehlersuche durch getrennte Testabteilung ist noch ärgerlicher (die sind zu doof zum Entwickeln und weisen mir permanent meine Fehlbarkeit nach)
- ☺ Programminspektion und seine Varianten sind u.a. ein Versuch, diese psychologischen Probleme in den Griff zu bekommen
- ☺ Rolle des Moderators ist von entscheidender Bedeutung für konstruktiven Verlauf von Inspektionen

Vorgehensweise bei der Programminspektion:

- **Inspektionsteam** besteht aus Moderator, Autor (passiv), Gutachter(n), Protokollführer und ggf. Vorleser (nicht dabei sind Vorgesetzte des Autors)
- **Gutachter** sind in aller Regel selbst (in anderen Projekten) Entwickler
- Inspektion **überprüft**, ob:
  - ⇨ Dokument Spezifikation erfüllt (Implementierung konsistent zu Modell)
  - ⇨ für Dokumenterstellung vorgeschriebene Standards eingehalten wurden
- Inspektion hat **nicht zum Ziel**:
  - ⇨ zu untersuchen, wie entdeckte Fehler behoben werden können
  - ⇨ Beurteilung der Fähigkeiten des Autors
  - ⇨ [lange Diskussion, ob ein entdeckter Fehler tatsächlich ein Fehler ist]
- **Inspektionsergebnis**:
  - ⇨ formalisiertes Inspektionsprotokoll mit Fehlerklassifizierung
  - ⇨ Fehlerstatistiken zur Verbesserung des Entwicklungsprozesses

Ablauf einer Inspektion:

- **Planung** des gesamten Verfahrens durch Management und Moderator
- **Auslösung** der Inspektion durch Autor eines Dokumentes (z.B. durch Freigabe)
- **Eingangsprüfung** durch Moderator (bei vielen offensichtlichen Fehlern wird das Dokument sofort zurückgewiesen)
- **Einführungssitzung**, bei der Prüfling den Gutachtern vorgestellt wird
- **Individualuntersuchung** des Prüflings (Ausschnitt) durch Gutachter (zur **Vorbereitung** auf gemeinsame Sitzung) anhand ausgeteilter Referenzdokumente
- auf **Inspektionssitzung** werden Prüfergebnisse mitgeteilt und protokolliert sowie Prüfling gemeinsam untersucht
- **Nachbereitung** der Sitzung und **Freigabe** des Prüflings durch Moderator (oder Rückgabe zur Überarbeitung)

(Technisches) Review (abgeschwächte Form der Inspektion):

- Prozessverbesserung und Erstellung von Statistiken steht nicht im Vordergrund
- Moderator gibt Prüfling nicht frei, sondern nur Empfehlung an Manager
- kein formaler Inspektionsplan mit wohldefinierten Inspektionsregeln
- Ggf. auch Diskussion alternativer Realisierungsansätze

Walkthrough (Informelles Review):

- Autor des Prüflings liest ihn vor (ablauforientiert im Falle von Software)
- Gutachter versuchen beim Vorlesen ohne weitere Vorbereitung Fehler zu finden
- Autor entscheidet selbst über weitere Vorgehensweise
- Zielsetzungen:
  - ⇨ Fehler/Probleme im Prüfling identifizieren
  - ⇨ Ausbildung/Einarbeitung von Mitarbeitern

3.4 Kontroll- und Datenflussorientierte Analysen

Der Kontrollflussgraph ist ... eine häufig verwendete Methode zur Darstellung von Programmen. ... Die Verarbeitungssteuerung übernehmen die Kontrollstrukturen der Software unter Nutzung der Datenwerte. Eingabedaten werden gelesen, um Zwischenergebnisse zu bestimmen, die in den Speicher geschrieben werden, ... Die Daten „fließen" quasi durch die Software; von Eingaben zu Ausgaben.

[Li02]

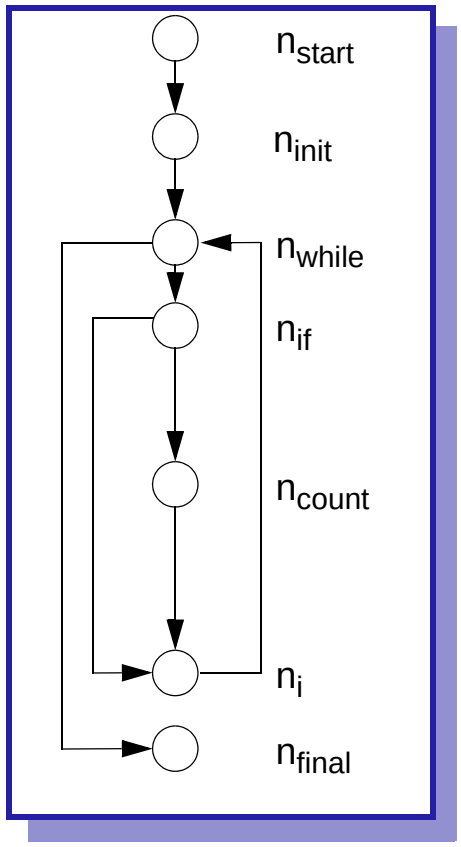
Definition „gerichteter Graph":

Ein **gerichteter Graph** G ist ein Tupel (N, E) mit

- ⇨ N := Menge von **Knoten** (Nodes)
- ⇨ $E \subseteq N \times N$ einer Menge gerichteter **Kanten**; eine Kante $e = (v_1 , v_2) \in E$ wird Kante von v_1 nach v_2 genannt

Kontrollflussgraph eines Programms (Prozedur, Methode):

```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);          (* start *)
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN

   count := 0; i := 0;          (* init *)

   WHILE s[i] # '.' DO

      IF    s[i]= 'a' OR s[i]= 'e' OR
            s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

            count := count+1;

      END;

      i := i+1;

   END (* WHILE *)
END countVowels;          (* final *)
```



Kontrollflussgraph
mit
- 7 **Knoten**
- 8 **Kanten**

Kontrollflussgraph - formale Definition:

Ein **Kontrollflussgraph** eines Programms (Prozedur) ist ein gerichteter Graph $G = (N, E, n_{start}, n_{final})$ mit

- ⇨ N ist die Menge der **Anweisungen** (Knoten = **N**odes) eines Programms
- ⇨ $E \subseteq N \times N$ ist die Menge der **Zweige** (Kanten = **E**dges) zwischen den Anweisungen des Programms
- ⇨ $n_{start} \in N$ ist der **Startknoten** des Programms
- ⇨ $n_{final} \in N$ ist der **Endknoten** des Programms (alternativ könnte man auch eine Menge von Endknoten betrachten)

Es gilt für den Kontrollflussgraphen:

- ⇨ $\neg\exists\, n \in N : (n, n_{start}) \in E$ - keine in den Startknoten einlaufenden Kanten
- ⇨ $\neg\exists\, n \in N : (n_{final}, n) \in E$ - keine aus dem Endknoten auslaufenden Kanten

Manchmal wird zusätzlich gefordert, dass Kontrollflussgraph zusammenhängend ist:

- ⇨ $\forall\, n \in N$: es gibt einen Pfad von n_{start} zu n
- ⇨ $\forall\, n \in N$: es gibt einen Pfad von n nach n_{final}

Pfade im Kontrollflussgraphen - formale Definition:

Ein **Pfad** in einem (Kontrollfluss-) Graphen $G = (N, E, n_{start}, n_{final})$ ist eine Knotenfolge $n_1, ... n_k$, sodass gilt:

- ⇨ $n_1, ... n_k \in N$
- ⇨ $\forall\ i \in 1, ... k\text{-}1 : (n_i , n_{i+1}) \in E$

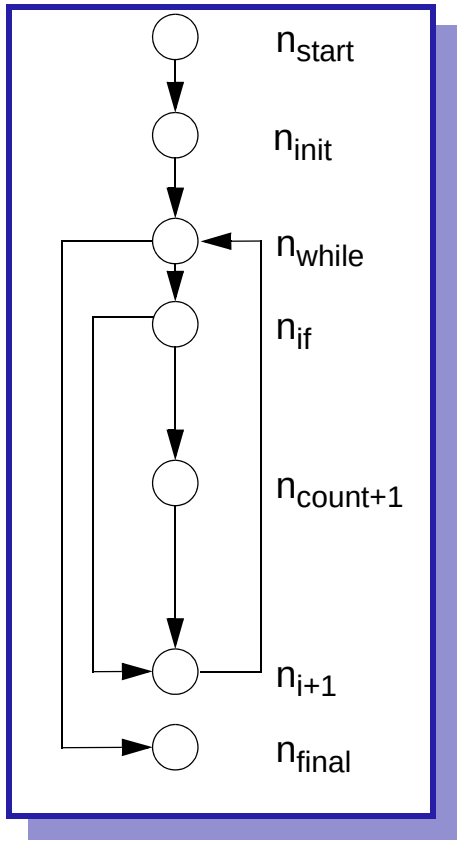
Ein **zyklenfreier Pfad** enthält keinen Knoten zweimal.

Ein **Zyklus** ist ein Pfad mit $n_1 = n_k$

Beispiel für einen Pfad:

n_{start} , n_{init} , n_{while} , n_{if} , n_{count} , n_i , n_{while} , ...

Der Pfad ist nicht zyklenfrei.



Kontrollflussgraph
mit
- 7 **Knoten**
- 8 **Kanten**

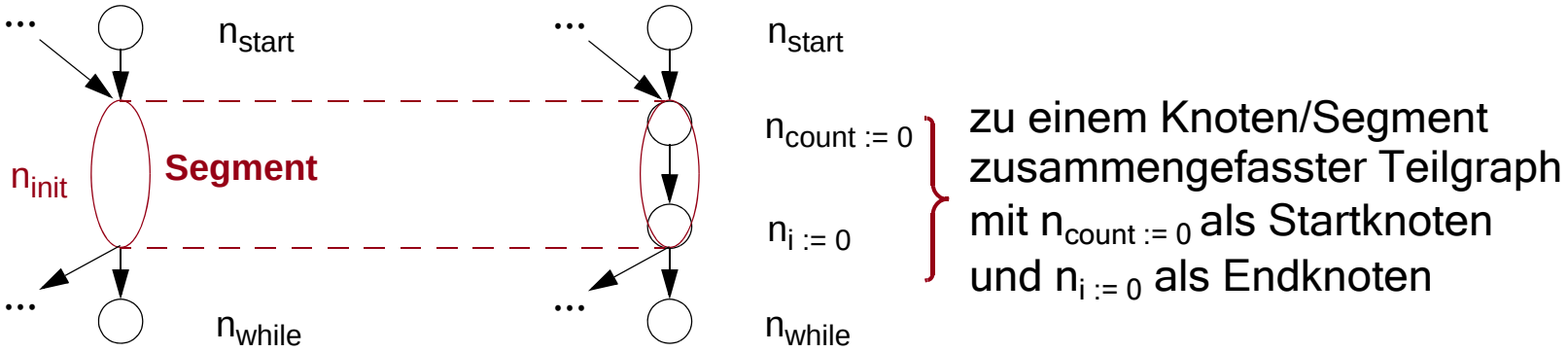
Kontrollflussgraphsegmente - formale Definition:

Ein **Segment** oder **Block** eines Kontrollflussgraphen $G = (N, E, n_{start}, n_{final})$ ist ein Knoten s, der einen Teilgraph $G' = (N', E', n'_{start}, n'_{final})$ von G ersetzt, der aus einem zyklenfreien Pfad $n'_{start} = n_1, \ldots, n_k = n'_{final}$ besteht mit $N' = \{n_1, \ldots, n_k\}$:

- $\Rightarrow$ $N' \subseteq N \wedge E' = E \cap (N' \times N') \wedge \neg (n'_{final}, n'_{start}) \in E'$
- $\Rightarrow$ $\forall n' \in N' \setminus \{n'_{final}\} \; \exists_1 n \in N : (n', n) \in E$ - genau eine auslaufende Kante
- $\Rightarrow$ $\forall n' \in N' \setminus \{n'_{start}\} \; \exists_1 n \in N : (n, n') \in E$ - genau eine einlaufende Kante

Einlaufende Kanten von n'_{start} und auslaufende von n'_{final} werden auf s umgelenkt.

Beispiel für ein Segment (kompakte Notation von Graphen):



Datenflussattribute eines Kontrollflussgraphen:

Die Anweisungen eines Kontrollflussgraphen G = (N, E, n_{start}, n_{final}) besitzen Datenflussattribute, die den Zugriffen auf Variable (Parameter, ...) in den Anweisungen entsprechen:

- ⇨ n ∈ N besitzt das Attribut **def(v)** oder kurz **d(v)**, falls
  n eine Zuweisung an v enthält (Wert von v definiert);
  gilt auch für Eingabeparameter bei n_{start}
- ⇨ n ∈ N besitzt das Attribut **c-use(v)** oder kurz **c(v)**, falls
  n eine Berechnung mit Zugriff auf v enthält (c = compute);
  implizite Zuweisung an Ausgabeparameter am Ende ist auch c-use
- ⇨ n ∈ N besitzt das Attribut **p-use(v)** oder kurz **p(v)**, falls
  n eine Fallentscheidung mit Zugriff auf v enthält (p = predicative)
- ⇨ n ∈ N besitzt das Attribut **r(v)**, falls es das Attribut **c(v)** oder **p(v)** besitzt, also lesend auf v zugreift (r = reference); dient nur der Zusammenfassung von c(v) und p(v), wenn Unterschied c/p irrelevant ist
- ⇨ n ∈ N besitzt das Attribut **u(v)**, falls v in dieser Anweisung (noch) keinen definierten Wert (mehr) besitzen kann

Kontrollflussgraph mit Datenflussattributen:

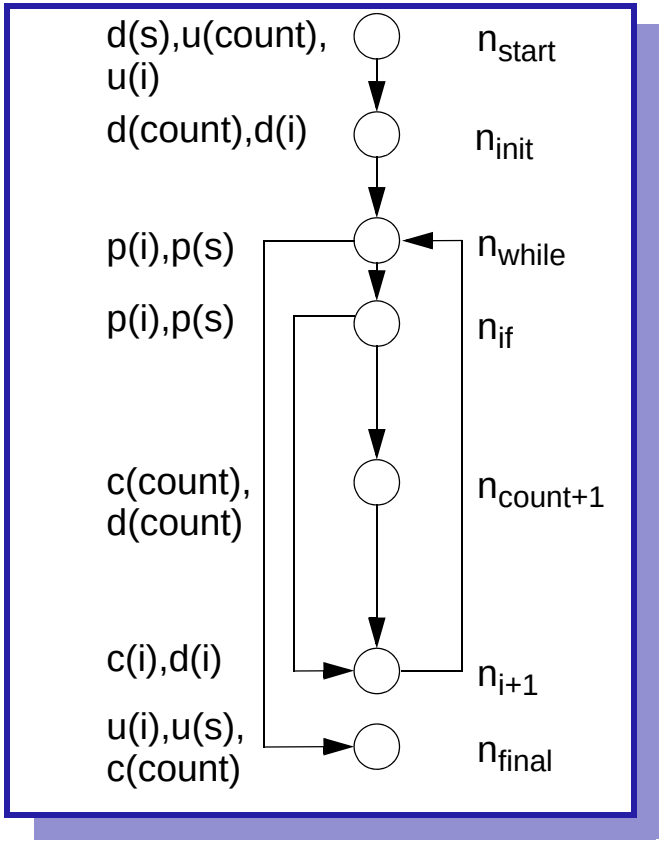
```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);          (* start *)
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN

   count := 0; i := 0;          (* init *)

   WHILE s[i] # '.' DO

      IF    s[i]= 'a' OR s[i]= 'e' OR
            s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

            count := count+1;

      END;

      i := i+1;

   END (* WHILE *)
END countVowels;          (* final *)
```

Achtung:

Reihenfolge der Datenfluss-attribute ist (manchmal) wichtig:

von links nach rechts (oben nach unten) lesen und auswerten.

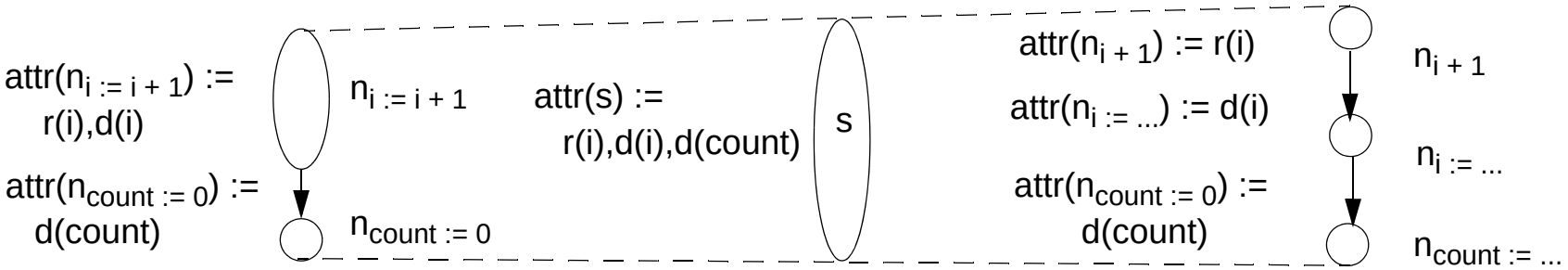
Mehrfache Auftreten von Datenfluss-attributen an einem Knoten werden meist zusammen-gefasst.



Kontrollflussgraphsegmente mit Datenflussattributen:

- Sei s ein Kontrollflussgraphsegment mit Pfad $n_{start} = n_1, \ldots, n_k = n_{final}$ und $attr(n_i)$ die Sequenz der Datenflussattribute der n_i . Dann ist $attr(s) := attr(n_1), \ldots, attr(n_k)$ die Konkatenierung der Datenflussattributsequenzen von n_1 bis n_k
- Ein Knoten n mit einer Sequenz von Datenflussattributen $attr(n) := attr_1, \ldots, attr_k$ ist immer eine abkürzende Schreibweise für einen Pfad von Knoten $n_1, \ldots, n_k$, sodass jeder Knoten n_i genau ein Datenflussattribut besitzt $attr(n_i) := attr_i$
- Allen folgenden Definitionen liegt immer ein „segmentfreier“ Kontrollflussgraph zugrunde, in dem jeder Knoten genau ein Datenflussattribut besitzt

Beispiel für Zusammenfassung/Expansion von Kontrollflussgraphknoten:


attr(n i := i + 1) := r(i),d(i)
attr(n count := 0) := d(count)
n i := i + 1
n count := 0
attr(s) := r(i),d(i),d(count)
s
attr(n i + 1) := r(i)
attr(n i := ...) := d(i)
attr(n count := 0) := d(count)
n i + 1
n i := ...
n count := ...

Werkzeug PMD - Kontrollflussgraph mit Datenflussattributen:

Line	Graph	Next nodes	Dataflow types
4	0	1	u(r)
4	1	2	
5	2	3	
7	3	4, 7	
9	4	5, 6	
10	5	4	d(r)
12	6	8	r(r)
14	7	8	d(r)
16	8	9	r(r)
17	9		u(r)

Type	Line(s)	Variable	Method
UR	4, 12	r	littleGauss
DD	10	r	littleGauss

Inout-Parameterdiskussion und Aufruf von Prozeduren:

Die meisten Programmiersprachen erlauben **nicht** die Auszeichnung von Variablen, die reinen Ausgabeparameter-Charakter besitzen; in Pascal/Modula gibt es nur mit VAR gekennzeichnete **inout-Parameter**, in Sprachen wie C oder C++ hat man nur die Möglichkeit, out-Parameter als Zeiger/Referenzen auf Variable/Objekte zu simulieren.

Für solche Parameter mit Ein- und Ausgabecharakter müssen wir wie folgt vorgehen:

- ⇨ Deklaration der Prozedur countVowels(IN s: ... ; INOUT count: ...):
  für n_{start} wird d(count) sowie d(s) angenommen, da beide Variablen bei der Übergabe einen definierten Wert haben sollten
  für n_{final} wird c(count) angenommen, da am Ende der Prozedur der Wert von count durch versteckte Zuweisung an Aufrufstelle übergeben wird
- ⇨ Aufruf der Prozedur countVowels(aSentence, aCounter):
  es wird c(aSentence) und c(aCounter) in normaler Anweisung oder p(aSentence) und p(aCounter) in Prädikat gefolgt von d(aCounter) angenommen, da beim Aufruf versteckte Zuweisungen die Werte von aSentence und aCounter an die Parameter s und count zuweisen

Datenflussgraph - formale Definition:

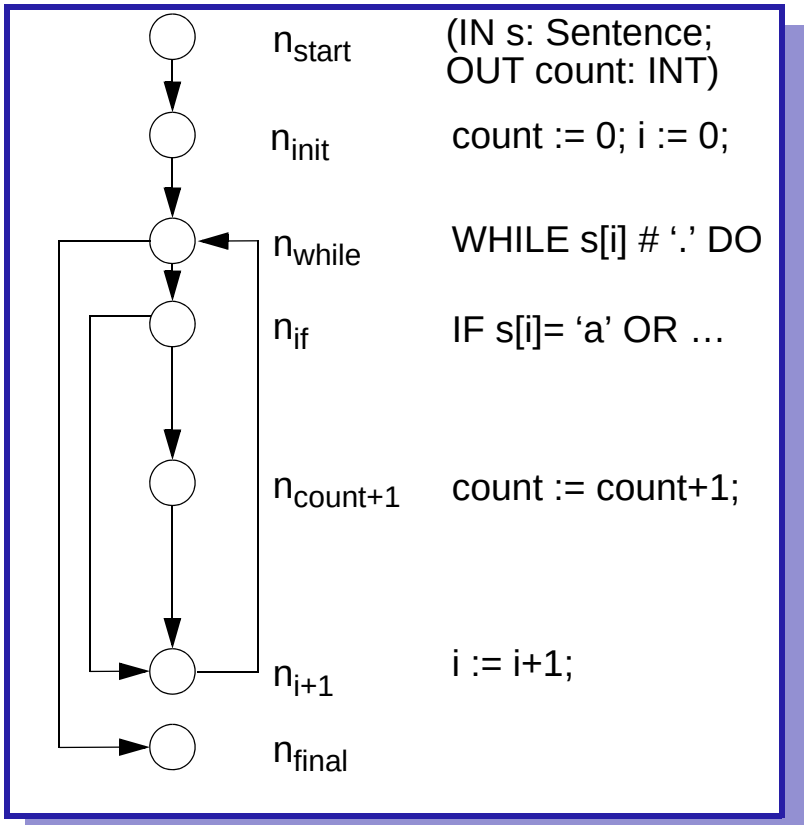
Ein **Datenflussgraph** $D = (V_d, E_d)$ zu einem Kontrollflussgraphen G eines Programms besteht aus

- ⇨ einer Menge von Knoten V_d für alle Anweisungen V des Programms
- ⇨ einer Menge von **Datenflusskanten** E_d :
  $(n_1, n_2) \in E_d$ genau dann, wenn es einen Pfad p im Kontrollflussgraphen G von n_1 nach n_2 gibt, sodass für eine Variable/Parameter v gilt:
  1. d(v) für n_1 : Anweisung n_1 definiert Wert für v
  2. r(v) für n_2 : Anweisung n_2 benutzt Wert von v
  3. für alle Anweisungen n auf Pfad p (ohne n_1) gilt nicht d(v) für n:
     n ändert also nicht den bei n_1 festgelegten Wert von v

Die Kanten des Datenflussgraphen verbinden also Zuweisungen an Variablen oder Parameter mit den Anweisungen, in denen die zugewiesenen Werte benutzt werden.

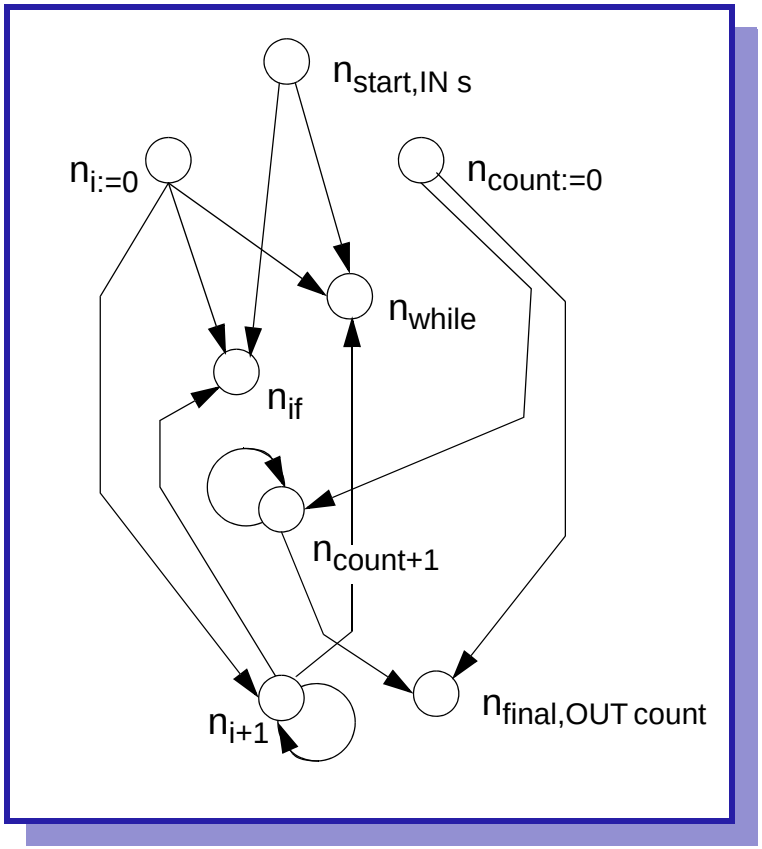
Beispiel für Datenflussgraph zu countVowels:

Kontrollflussgraph (countVowels):



Knoten = Anweisungsblöcke
Kanten = Kontrollfluss

Datenflussgraph (countVowels):



Knoten = setzende/lesende Variablenzugriffe
Kanten = Datenfluss der zugewiesenen Werte

Programm-Slices zur Fehlersuche und Änderungsfolgeschätzung:

Ein **Abhängigkeitsgraph** $A = (N_a, E_a)$ eines Programms ist ein gerichteter Graph, der

- ⇨ alle Knoten und Kanten des Datenflussgraphen enthält (ohne Zusammenfassung von Teilgraphen zu Segmenten) sowie zusätzlich
- ⇨ Kanten für Pfade des Kontrollflussgraphen von allen Bedingungen zu direkt kontrollierten Anweisungen (das sind die Anweisungen, deren Ausführung von der Auswertung der betrachteten Bedingung direkt abhängt).

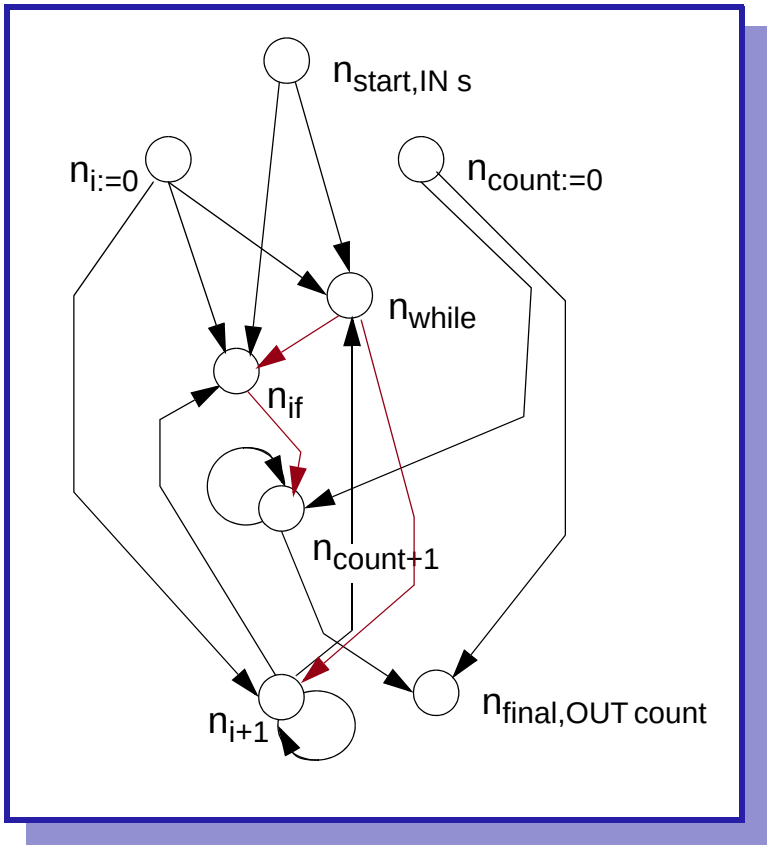
Es gibt zwei Arten von Ausschnitten (Slices) eines Abhängigkeitsgraphen A:

- ❑ **Vorwärts-Slice** für Knoten $n \in N_a$ mit Datenflussattribut d(v):
  alle Pfade in A, die von Knoten n ausgehen, der Variable v definiert
  (der Slice-Graph enthält alle Knoten und Kanten der Pfade)
- ❑ **Rückwärts-Slice** für Knoten $n \in N_a$ mit Datenflussattribut r(v):
  alle Pfade in A, die in Knoten n einlaufen, der Variable v referenziert
  (der Slice-Graph enthält alle Knoten und Kanten der Pfade)

Abhängigkeitsgraph zu countVowels:

```
PROCEDURE countVowels(IN s: Sentence;
                      OUT count: INTEGER);

VAR i: INTEGER;

BEGIN

   count := 0; i := 0;        (* init *)

   WHILE s[i] # '.' DO

      IF    s[i]= 'a' OR s[i]= 'e' OR
            s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

            count := count+1;

      END;

      i := i+1;

   END (* WHILE *)
END countVowels;        (* final *)
```

Slice für alle Knoten von countVowels:



schwarze Kanten: Datenflusskanten
rote Kanten: Kontrollflusskanten

Vorwärts-Slice - Beispiel und Nutzen:

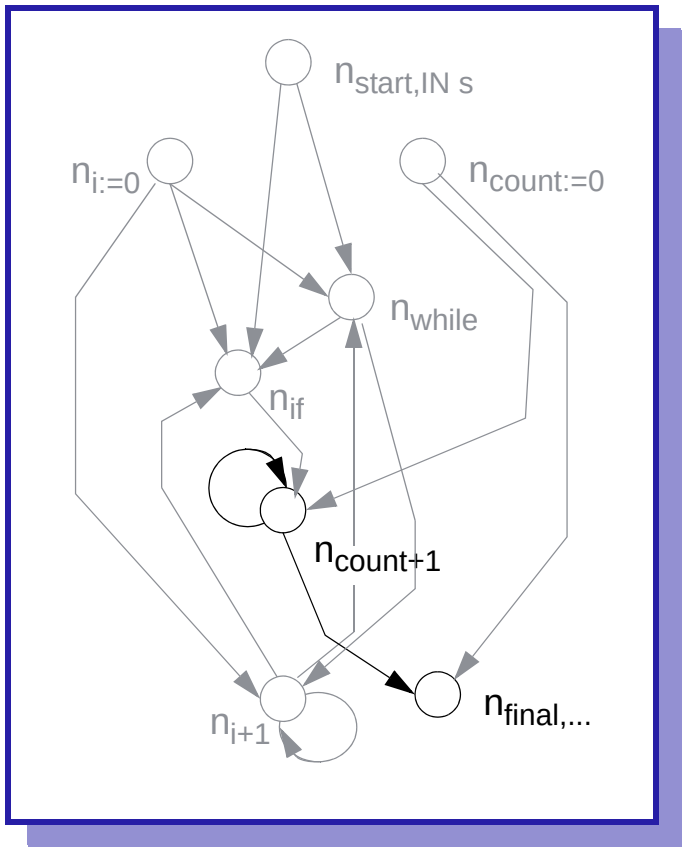
Ein **Vorwärts-Slice** zu einer Anweisung n, die einer Variable v einen Wert zuweist, bestimmt alle die Stellen eines Programms, die von einer Änderung des zugewiesenen Wertes (Berechnungsvorschrift) betroffen sein könnten.

Der Vorwärts-Slice zu count := count+1:

```
...
        count := count+1;
...
END countVowels;      (* final mit Rückgabe count *)
```

Ein Vorwärts-Slice dient der Abschätzung von Folgen einer Programmänderung.

Vorwärts-Slice für count := count+1:



Rückwärts-Slice - Beispiel und Nutzen:

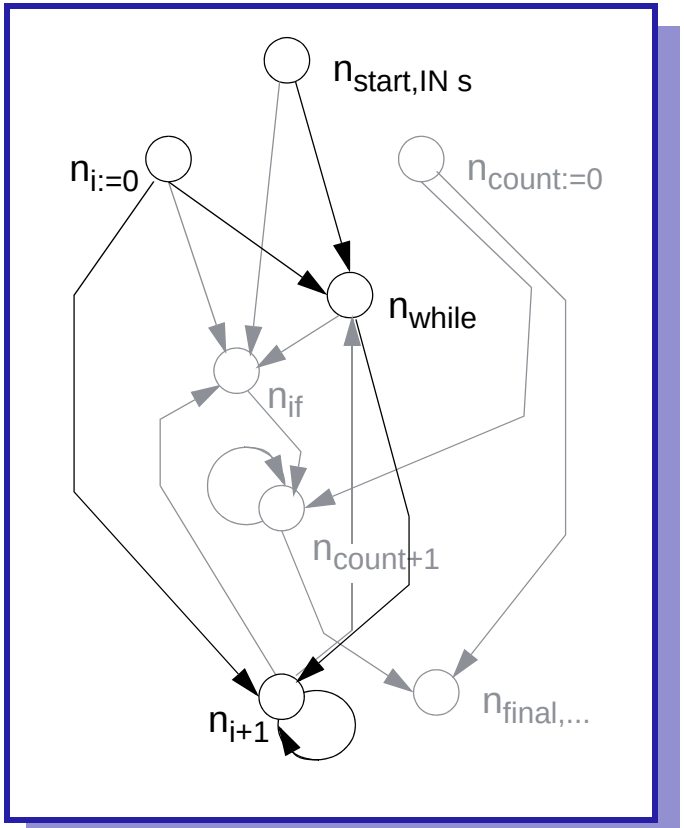
Ein **Rückwärts-Slice** zu einer Anweisung n, die eine Variable referenziert, bestimmt alle die Stellen eines Programms, die den Wert der Variable direkt oder indirekt bestimmt haben.

Der Rückwärts-Slice zu i := i+1:

```
PROCEDURE countVowels(  IN s: Sentence;
                        OUT count: INTEGER);
   VAR i: INTEGER;
BEGIN                   (* start *)
   count := 0; i := 0;
   WHILE s[i] # '.' DO
      ...
      i := i+1;
```

Ein Rückwärts-Slice ist bei der Fehlersuche hilfreich, um schnell irrelevante Programmteile ausblenden zu können.

Rückwärts-Slice für i := i+1:



Datenfluss- und Kontrollflussanomalien:

Eine **Anomalie** eines Programms ist eine verdächtige Stelle in dem Programm. Eine solche verdächtige Stelle ist keine garantiert fehlerhafte Stelle, aber eine potentiell fehlerhafte Stelle im Programm.

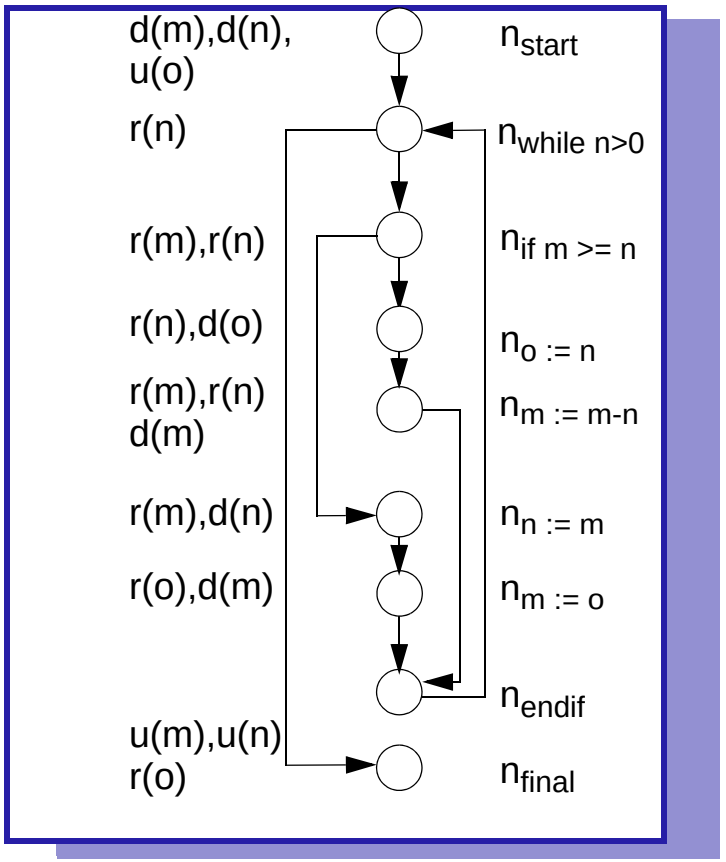
Datenflussanomalien (meist deutliche Hinweise auf Fehler) sind etwa:

- ⇨ es gibt einen Pfad im Kontrollflussgraphen, auf dem eine Variable v referenziert wird bevor sie zum ersten Mal definiert wird (Zugriff auf undefinierte Variable)
- ⇨ es gibt einen Pfad im Kontrollflussgraphen, auf dem eine Variable v zweimal definiert wird ohne zwischen den Definitionsstellen referenziert zu werden (nutzlose Zuweisung an Variable)

Kontrollflussanomalien sind bei modernen Programmiersprachen von geringerer Bedeutung. Im wesentlichen handelt es sich dabei bei Programmiersprachen ohne „goto“-Anweisungen um nicht erreichbaren Code (ansonsten beispielsweise Sprünge in Schleifen hinein).

Beispiel für Programm mit Datenflussanomalien:

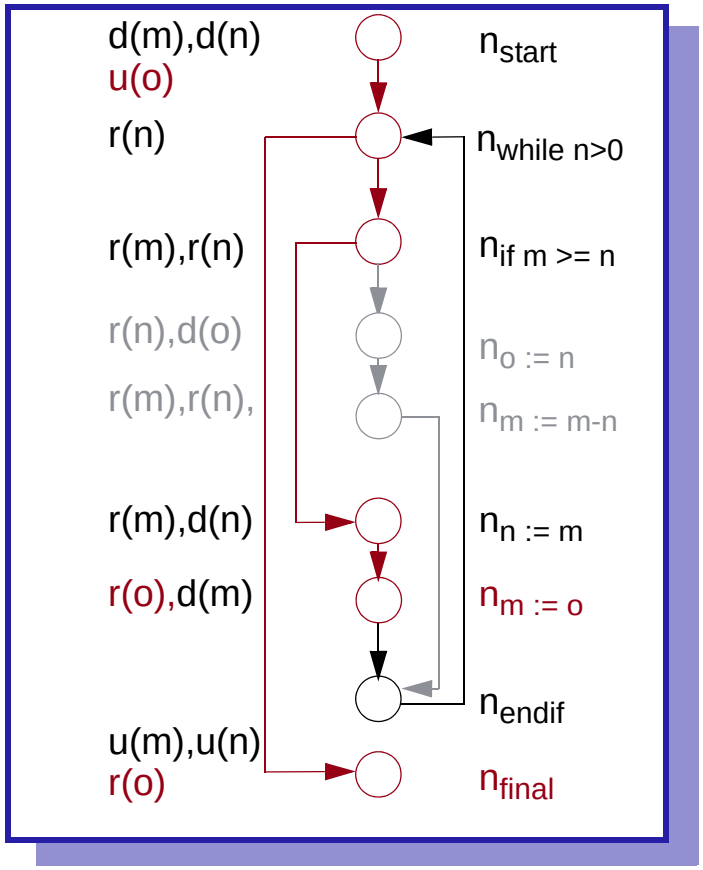
```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);        (* start *)
BEGIN
   WHILE n>0 DO
      IF m >= n  THEN
         o := n;
         m := m-n;
      ELSE
         n := m;
         m := o;
      END;                                              (* endif *)
END ggT;                                                (* final *)
```



Undefined-Reference-Datenflussanomalie:

Eine **ur-Datenflussanomalie** bezüglich einer Variable v ist wie folgt definiert:

- ⇨ es gibt einen segmentfreien Pfad n_1, ... n_k
- ⇨ n_1 hat Attribut u(v), v besitzt also keinen definierten Wert bei n_1
- ⇨ n_2, ... , n_{k-1} hat nicht Attribut d(v), v erhält also keinen definierten Wert
- ⇨ n_k hat Attribut r(v), auf v wird also lesend zugegriffen



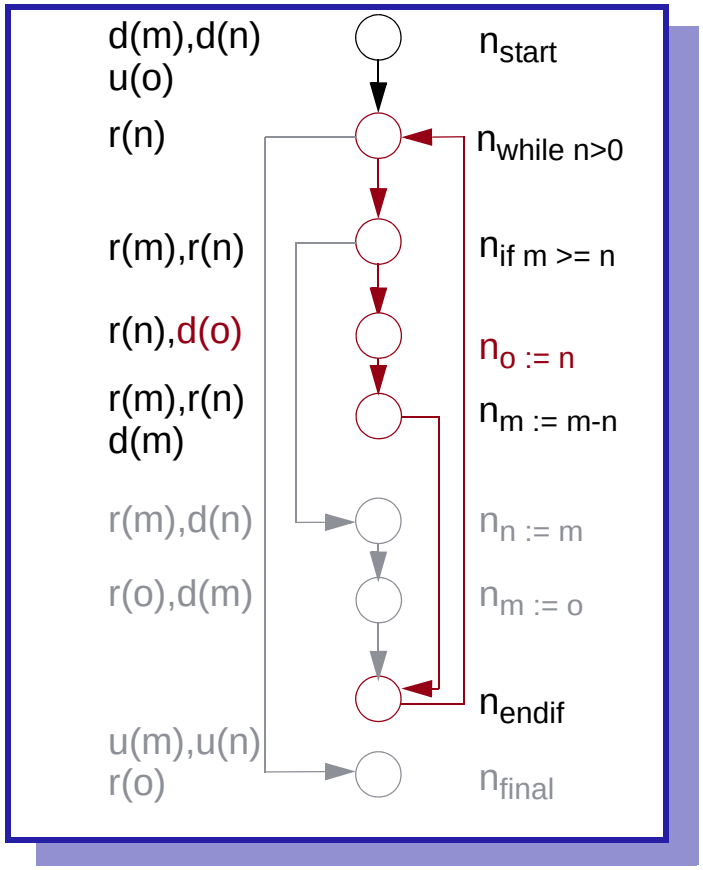
Beispiele für ur-Datenflussanomalien:

- ❑ Aufruf von ggT mit m = n = 0: bei n_{final} besitzt o keinen Wert
- ❑ Aufruf von ggT mit m = 2 und n = 6 bei $n_{m := o}$ besitzt o keinen Wert

Defined-Defined-Datenflussanomalie:

Eine **dd-Datenflussanomalie** bezüglich einer Variable v ist wie folgt definiert:

- ⇨ es gibt einen segmentfreien Pfad n_1, ... n_k
- ⇨ n_1 hat Attribut d(v), v erhält also bei n_1 einen neuen Wert
- ⇨ n_2, ... , n_{k-1} hat nicht Attribut [r|d|u](v), v wird also bis n_k nicht verwendet
- ⇨ n_k hat Attribut d(v), alter Wert von v wird bei n_k unbenutzt überschrieben



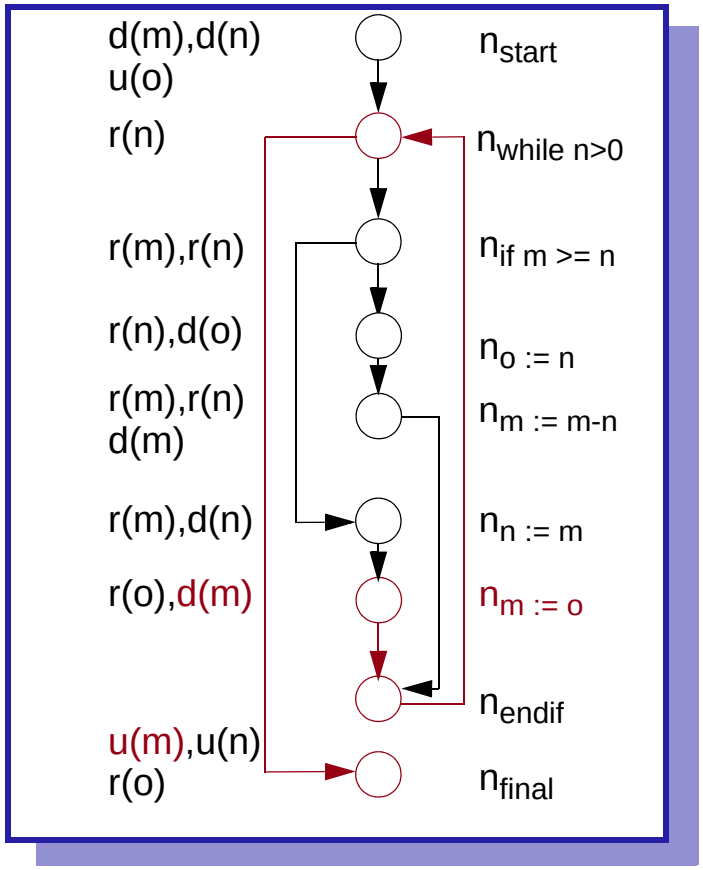
Beispiel für dd-Datenflussanomalie:

- ❑ Aufruf von ggT mit m = 6 und n = 2:
  bei $n_{o := n}$ wird Wert von o beim zweiten Mal überschrieben, ohne vorher referenziert zu werden.

Defined-Undefined-Datenflussanomalie:

Eine **du-Datenflussanomalie** bezüglich einer Variable v ist wie folgt definiert:

- ⇨ es gibt einen segmentfreien Pfad $n_1, \ldots n_k$
- ⇨ n_1 hat Attribut d(v), v erhält also einen definierten Wert bei n_1
- ⇨ $n_2, \ldots, n_{k-1}$ hat nicht Attribut [r|d|u](v), v wird also bis n_k nicht verwendet
- ⇨ n_k hat Attribut u(v), v wird also auf undefiniert gesetzt

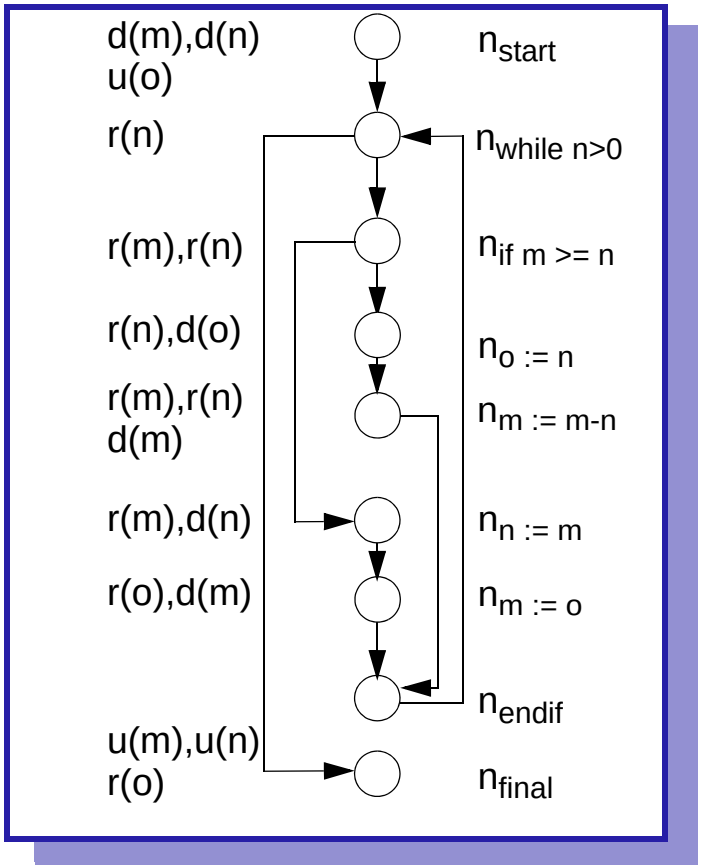


Beispiel für du-Datenflussanomalie:

- ❑ Aufruf von ggT mit m = 2, n = 2:
  bei $n_{m := o}$ erhält m definierten Wert, der nie mehr referenziert wird

Korrektur des Programms mit Datenflussanomalien - geht das?

```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);        (* start *)
BEGIN
   WHILE n>0 DO
      IF m >= n  THEN
         o := n;              (* dd für o *)
         m := m-n;            (* du für m *)
      ELSE
         n := m;
         m := o;   (* ur für o, du für m *)
      END;                    (* endif *)
END ggT;                      (* final *)
                              (* ur für o *)
```



Korrigiertes ggT-Programm mit verbleibender du-Datenflussanomalie:

```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);          (* start *)
BEGIN
  o := n;
  WHILE n > 0 DO
    IF m >= n  THEN
      m := m-n (**)
    ELSE
      n := m;
      if n > 0 THEN
        m := o; (***)
        o := n;
      END
  END
END ggT;                                                  (* final *)
```

- die Anweisung „m := o;“ führt statisch gesehen zu einer du-Anomalie, da while-Schleife nach Zuweisung beendet sein könnte
- tatsächlich passiert aber zur Laufzeit das:
  - ⇨ wenn m ein neuer Wert bei (***) zugewiesen wird, muss n größer als 0 sein
  - ⇨ dann wird die while-Schleife wieder betreten und auf m lesend zugegriffen
  - ⇨ es gibt also in Realität keine nutzlose Zuweisung an m
- weitere scheinbare du-Anomalie wird durch die Zuweisung (**) hervorgerufen
- echte du-Anomalie: auf m wird für n = 0 nie lesend zugegriffen

Probleme mit statischer Datenflussanalyse für Datenstrukturen:

- ☹ funktioniert nicht (gut) für komplexe Datenstrukturen wie Felder (Arrays), bei denen man jede einzelne Komponente wie eigene Variable behandeln müsste:

```
s[i] := x;
y := s[k];
```

  Obiges Programmfragment hat keine du- und ur-Anomalie, falls i = k ; aber die Gleichheit von i und k (oder anderen Ausdrücken) lässt sich im allgemeinen nur schwer garantieren.

- ☹ noch größere Probleme hat man bei verzeigerten Datenstrukturen:

```
delete(obj1);
obj3 := obj2;
```

  Obiges Programmfragment ist nur dann in Ordnung, wenn die Variablen obj1 und obj2 nicht auf dasselbe Objekt zeigen (sonst würde nämlich in der zweiten Zeile u(obj1) und u(obj2) gelten).

- ☹ Unterscheidung von in-, out, und inout-Parametern (in Java, C++, ...)

Probleme mit statischer Datenflussanalyse bei Fallunterscheidungen:

```
IF Bed1 THEN x := expr1 ELSE y := expr2 END;

IF Bed2 THEN z := x ELSE z := y END;
```

Das obige Programm funktioniert, falls Bed1 und Bed2 äquivalent sind.
Trotzdem liefert die Datenflussanalyse immer Anomalien, da von ihr die Äquivalenz von Bed1 und Bed2 nicht erkannt wird.

Problem:

- ⇨ reale Programme enthalten oft viele Anomalien, die nicht echte Programmierfehler sind (zu viele nutzlose Warnungen werden erzeugt)
- ⇨ restriktivere Definitionen von sogenannten starken Anomalien übersehen andererseits u.U. zu viele echte Fehler (siehe folgende Folien)

Lösung:

- ⇨ zunächst neue Definitionen **„starker“ Anomalien** verwenden
- ⇨ dann bisherige Definitionen von **(schwachen) Anomalien** verwenden

Definitionen starker Datenflussanomalien:

Geg. Kontrollflussgraph G zu Programm P mit Datenflussattributen (ohne Segmente):

- **starke ur-Anomalie**: zu Anweisungen n mit Attribut u(v) und über einen Pfad von n erreichbarem n' mit r(v) gibt es **keinen** segmentfreien Pfad in G $n = n_1 , ... , n_k = n'$ in dem n' nur einmal auftritt und für den gilt:

  es existiert $i \in 2, ... , k-1$ mit: n_i besitzt Attribut d(v) oder u(v)

  Idee dieser Definition:

  - ⇨ von n mit u(v) nach n' mit r(v) gibt es **mindestens** einen Ausführungspfad
  - ⇨ ausgeschlossen werden **zyklische Pfade** durch n', um so die Analyse auf die erste Ausführung einer Anweisung in einer Schleife zu einzuschränken
  - ⇨ auf **keinem** Pfad wird an Variable v ein Wert zugewiesen, bevor bei n' lesend auf v zugegriffen wird
  - ⇨ des weiteren werden Situationen ausgeschlossen, bei denen die gerade betrachtete ur-Anomalie (teilweise) durch irgendeine andere Anomalie „**überlagert**“ wird

Definitionen starker Datenflussanomalien - Fortsetzung:

- **starke du-Anomalie**: zu Anweisungen n mit Attribut d(v) und über einen Pfad von n erreichbarem n' mit u(v) gibt es **keinen** segmentfreien Pfad in G $n = n_1 , ... , n_k = n'$ in dem n' nur einmal auftritt und für den gilt:

  es existiert $i \in 2, ... , k-1$ mit: n_i besitzt Attribut d(v) oder u(v) oder r(v)

  Idee dieser Definition:

  - ⇨ von n mit d(v) nach n' mit u(v) gibt es **mindestens** einen Ausführungspfad
  - ⇨ ausgeschlossen werden wieder **zyklische Pfade**, um so die Analyse auf die erste Ausführung einer Anweisung in einer Schleife zu einzuschränken
  - ⇨ auf **keinem** Pfad wird an Variable v bei n zugewiesener Wert verwendet, bevor er bei n' undefiniert wird
  - ⇨ des weiteren werden Situationen ausgeschlossen, bei denen die gerade betrachtete du-Anomalie (teilweise) durch irgendeine andere Anomalie „**überlagert**“ wird

Definitionen starker Datenflussanomalien - Fortsetzung:

- **starke dd-Anomalie**: zu Anweisungen n mit Attribut d(v) und über einen Pfad von n erreichbarem n' mit d(v) gibt es **keinen** segmentfreien Pfad in G $n = n_1 , ... , n_k = n'$ in dem n' nach n_1 nur einmal auftritt und für den gilt

  es existiert $i \in 2, ... , k-1$ mit: n_i besitzt Attribut d(v) oder u(v) oder r(v)

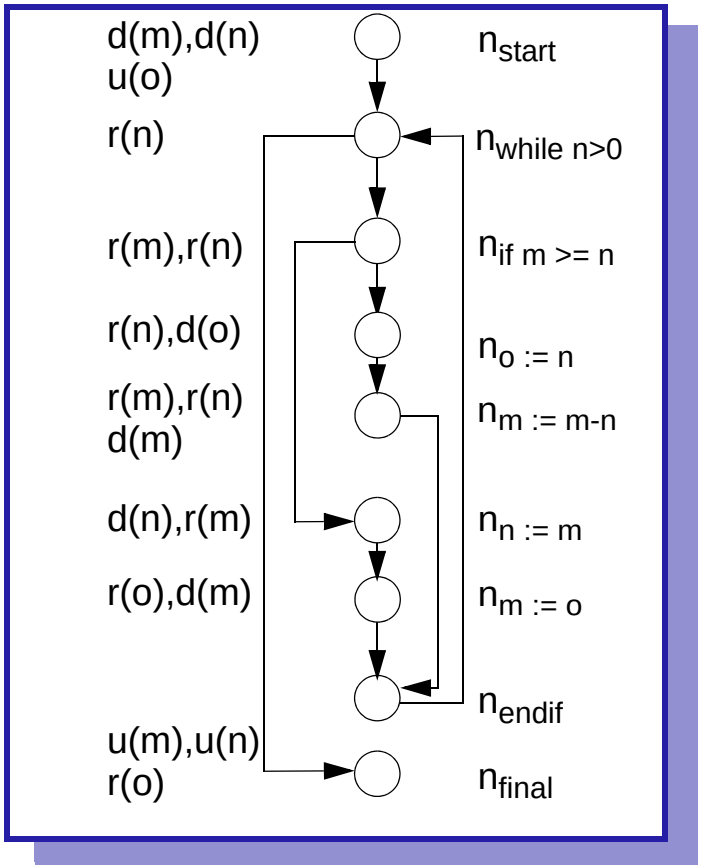
  Idee dieser Definition:

  - ⇨ von n mit d(v) nach n' mit d(v) gibt es **mindestens** einen Ausführungspfad
  - ⇨ ausgeschlossen werden wieder **zyklische Pfade**, um so die Analyse auf die erste Ausführung einer Anweisung in einer Schleife zu einzuschränken
  - ⇨ auf **keinem** Pfad wird an Variable v bei n zugewiesener Wert verwendet, bevor bei n' erneut ein Wert zugewiesen wird
  - ⇨ des weiteren werden Situationen ausgeschlossen, bei denen die gerade betrachtete dd-Anomalie (teilweise) durch irgendeine andere Anomalie „**überlagert**" wird

Achtung: unser Programm ggT enthält keine starken Datenflussanomalien, sondern ausschließlich potentielle Datenflussanomalien!

Beispiel für Programm ohne starke Datenflussanomalien:

```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);          (* start *)
BEGIN
   WHILE n>0 DO
      IF m >= n  THEN
         o := n; (* keine starke dd-Anomalie *)
         m := m-n;
      ELSE
         n := m;
         m := o; (* keine starke ur-Anomalie *)
      END;                                                (* endif *)
END ggT;                                                  (* final *)
(* keine starke du-Anomalie für m *)
```



3.5 Softwaremetriken

Die Definition von Software-Maßen basiert auf dem Wunsch, einen quantitativen Zugang zum abstrakten Produkt Software zu gewinnen. Dabei ist zwischen der Vermessung von Eigenschaften einer Software und der quantitativen Kontrolle des zugrundeliegenden Entwicklungs-prozesses zu unterscheiden. [Li02]

- **Produktmetriken** messen Eigenschaften der Software:
  - ⇨ Qualität der Software (z.B. als Anzahl gefundener Fehler)
  - ⇨ Einhaltung von Standards (z.B. als Anzahl Verletzung von Stilregeln)
- **Prozessmetriken** messen Eigenschaften des Entwicklungsprozesses:
  - ⇨ Dauer oder Kosten der Entwicklung (z.B. als Mitarbeitermonate)
  - ⇨ Zufriedenheit des Kunden (z.B. als Anzahl Änderungswünsche)?

Achtung: die Software-Engineering-Literatur verwendet meist die Begriffe „Maß“ und „Metrik“ als Synonyme (Metrik in der Mathematik: Entfernung zweier Punkte, die gemessen werden kann)

Gewünschte Eigenschaften von Maß/Metrik:

- **Einfachheit**: berechnetes Maß lässt sich einfach interpretieren (z.B. Zeilenzahl einer Datei)
- **Eignung** (Validität): es besteht ein (einfacher) Zusammenhang zwischen der gemessenen Eigenschaft und der interessanten Eigenschaft (z.B. zwischen Programmlänge und Fehleranzahl)
- **Stabilität**: gemessene Werte sind stabil gegenüber Manipulationen untergeordneter Bedeutung (z.B. die Unterschiede zwischen zwei Projekten, wenn man aus erstem Projekt Rückschlüsse auf zweites Projekt ziehen will)
- **Rechtzeitigkeit**: das Maß kann zu einem Zeitpunkt berechnet werden, zu dem es noch zur Steuerung des Entwicklungsprozesses hilfreich ist (Gegenbeispiel: Programmlänge als Maß für Schätzung des Entwicklungsaufwandes)
- **Reproduzierbarkeit**: am besten automatisch berechenbar ohne subjektive Einflussnahme des Messenden (Gegenbeispiel: Beurteilung der Lesbarkeit eines Programms durch manuelle Durchsicht)

Maßskalen:

- **Nominalskala**: frei gewählte Menge von Bezeichnungen wie etwa
  Programm in C++, Java, Fortran, ... geschrieben
- **Ordinalskala**: geordnete Menge von Bezeichnern wie etwa
  Programm gut lesbar, einigermaßen lesbar, ... , absolut grauenhaft
- **Rationalskala**: Messwerte können zueinander in Relation gesetzt werden und prozentuale Aussagen mit Multiplikation und Division sind sinnvoll wie etwa
  Programm A besitzt doppelt/halb soviele Programmzeilen wie Programm B

  weiteres Beispiel:

Zielsetzung von Softwaremetriken hier:

- Metrik soll zur **Prognose der Fehler** (einer bestimmten Art) in einem Softwaresystem eingesetzt werden
- **Softwarequalität** wird also mit etwas „leicht“ messbaren wie der Anzahl der Fehler pro Codezeile (die bis zum Zeitpunkt x gefunden wurden) gleichgesetzt
- **1. Hypothese:** komplexer Code enthält mehr Fehler als einfacher Code (pro Zeile Quelltext)
- gesucht werden also **Metriken für Komplexität** eines untersuchten (gemessenen) Softwaresystems
- **2. Hypothese**: es gibt einfachen Zusammenhang zwischen der gemessenen Softwarekomplexität und der Anzahl später gefundener Fehler (pro Codezeile)

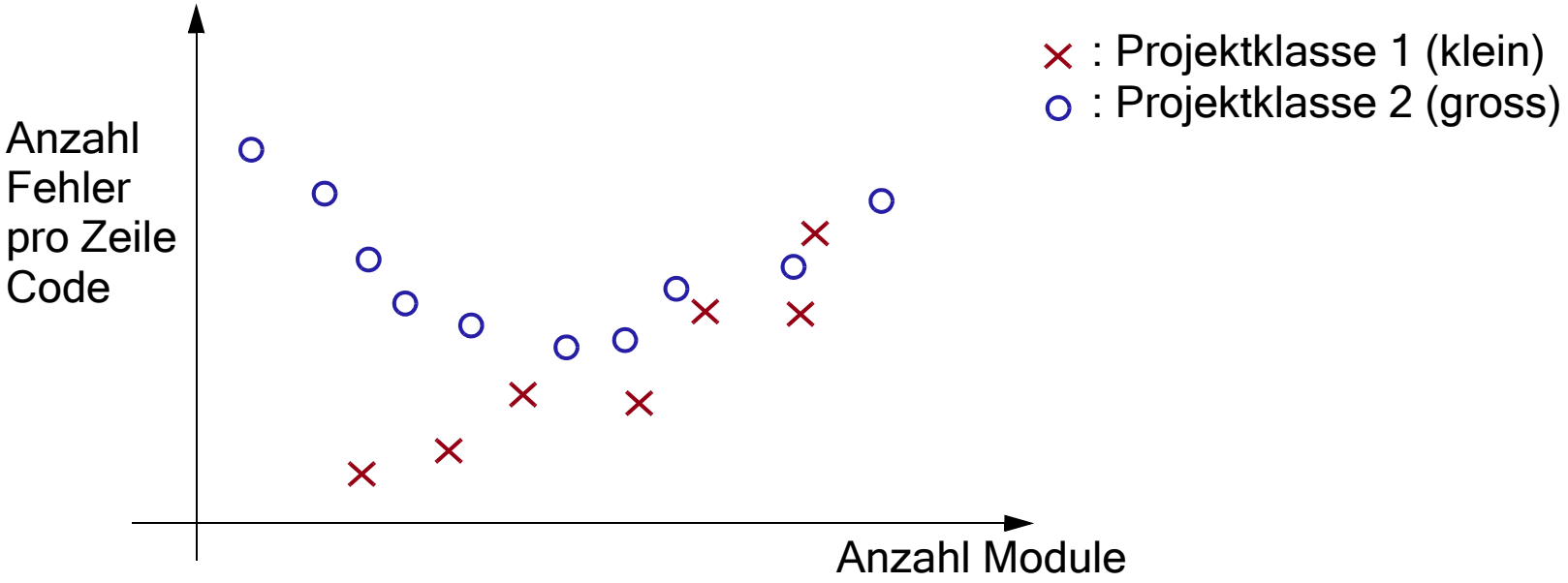
Frage:

Wie „validiert“ man all diese Hypothesen???

Beispiel für eine falsche Hypothese:

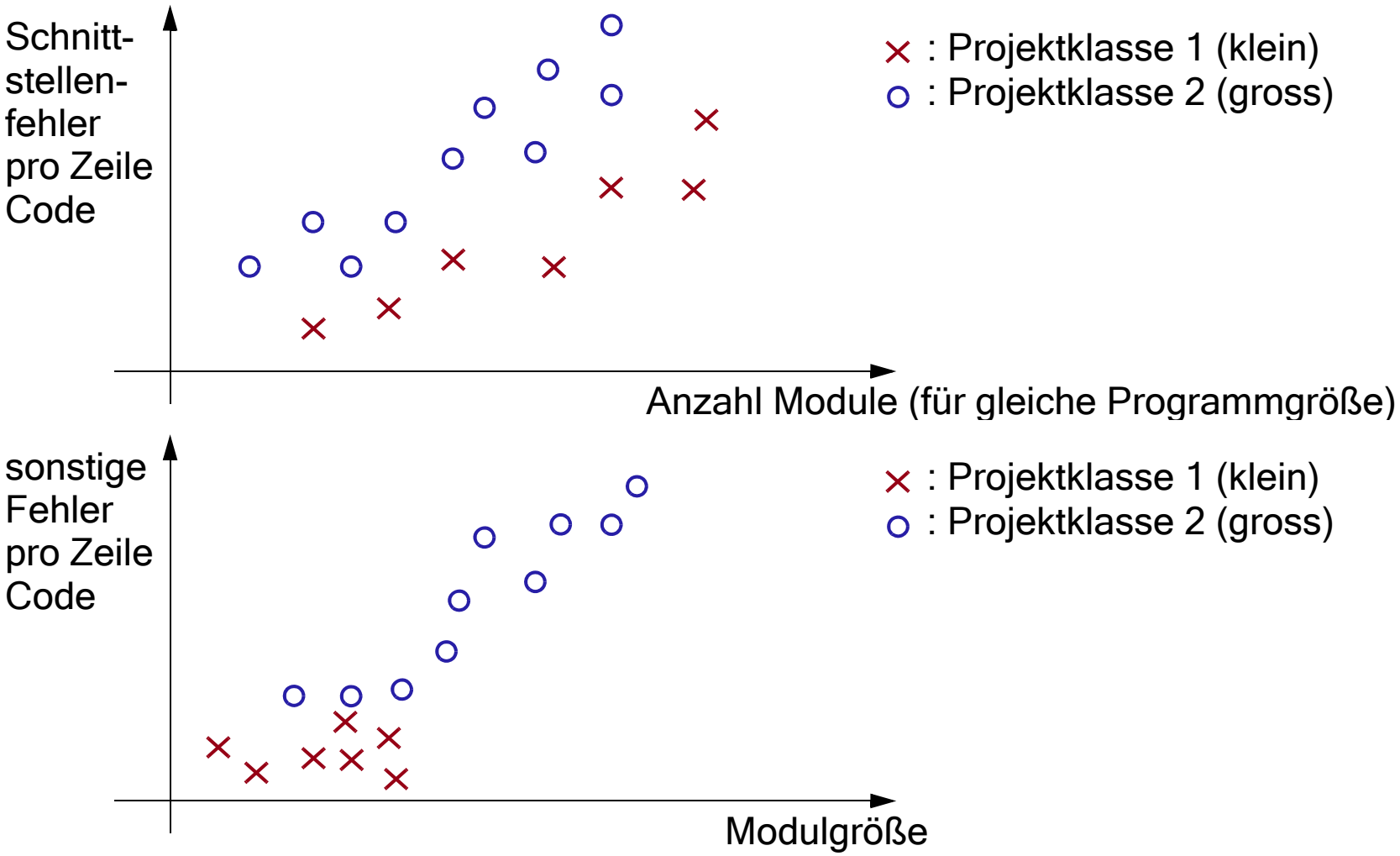
Modularisierung von Programmen verringert die Zahl der Fehler, also steigende Anzahl von Modulen verursacht fallende Fehlerzahl.

Gemessene Werte für verschiedene Projekte:

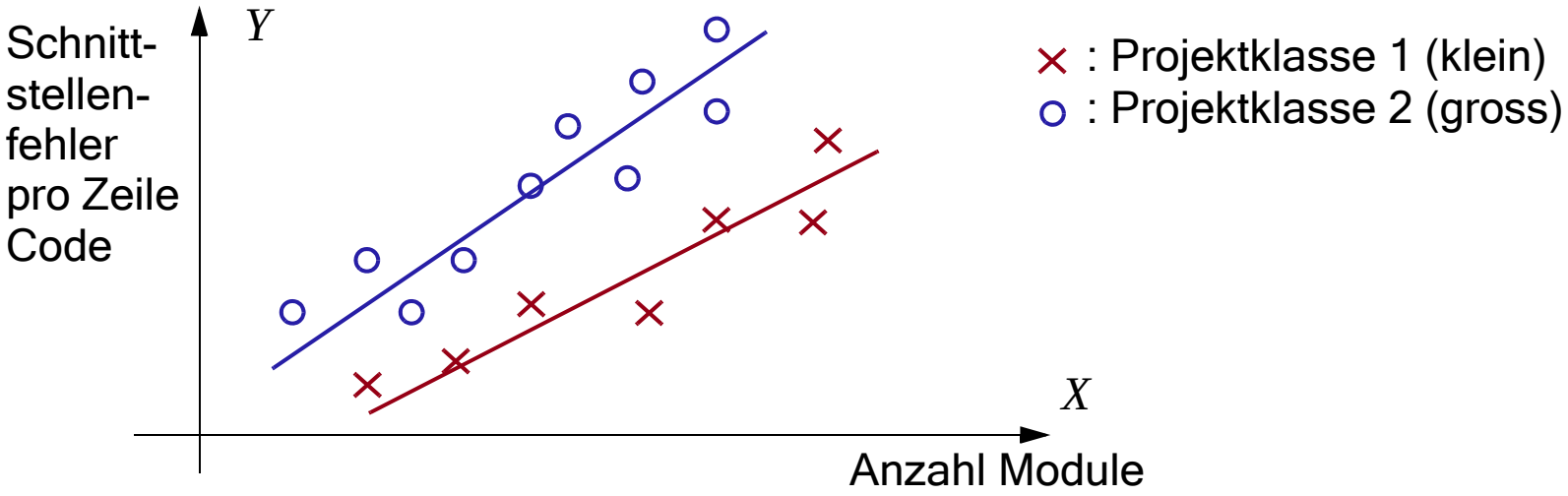


Die hier aufgetragenen Werte sind fiktiv, reale Untersuchungen haben aber qualitativ ähnliche Ergebnisse geliefert.

Eine Erklärung für tatsächlichen Zusammenhang Modulzahl - Fehlerzahl:



Testen von Hypothesen mit Regressionsrechnung:



Getestet werden soll die Hypothese, dass ein linearer Zusammenhang zwischen der Anzahl der Module und der Anzahl der Schnittstellenfehler pro Zeile Code in einem Programm besteht.

1. es wird die Gerade ermittelt, die die Messwerte nach der Methode der kleinsten Fehlerquadrate am besten approximiert (nach Gauss)

2. es wird berechnet, wie gut die Streuung der Messwerte in *Y*-Richtung durch zufällig verteilte Messfehler um berechnete Gerade herum erklärt wird

Berechnung der Regressionsgeraden:

Gesucht wird: $Y = b_0 + b_1 X$

Gegeben sind Paare von Messwerten: (x_1,y_1) , ... , (x_n,y_n)

Berechnung der Mittelwerte:

Mittelwert $\bar{x} = (x_1 + ... + x_n) / n$

Mittelwert $\bar{y} = (y_1 + ... + y_n) / n$

Berechnung von Koeffizient b_1 :

$$b_1 = \frac{\frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Berechnung von Koeffizient b_0 :

$$b_0 = \bar{y} - b_1 \bar{x}$$

Berechnung des Korrelationskoeffizienten r:

$$r = \frac{\sum_{i=1}^{n} (x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2 \cdot \sum_{i=1}^{n} (y_i - \bar{y})^2}}$$

- man kann zeigen, dass der **Korrelationskoeffizient** $r \in [-1 .. +1]$ gilt
- die Grenzfälle $r = +1$ und $r = -1$ treten auf, wenn schon alle gemessenen Punkte (x_i, y_i) auf einer Geraden liegen
- die Regressionsgerade steigt für $r = +1$ und fällt für $r = -1$
- für $r = 0$ verläuft die Gerade parallel zur X-Achse, es besteht also kein (linearer) Zusammenhang zwischen X- und Y-Werten
- r^2 heisst **Bestimmtheitsmaß** und lässt sich interpretieren als Anteil der durch die Regression erklärten Streuung der Y-Werte
- hat man z. B. $r = 0.7$ erhalten, dann ist $r^2 = 0.49$, d.h. 49 % der Streuung der Y-Werte werden durch die lineare Abhängigkeit von X erklärt

Auswertung von ordinalen/rationalen Metriken:

1. aufgrund von „Erfahrungswerten“ sind sinnvolle untere und obere Grenzwerte für einen Messwert bekannt (siehe Kiviatdiagramm)

   ⇨ alle Komponenten (Module, Klassen, Methoden, ...) mit kritischen Werten werden genauer untersucht und ggf. „saniert“ (neu geschrieben)

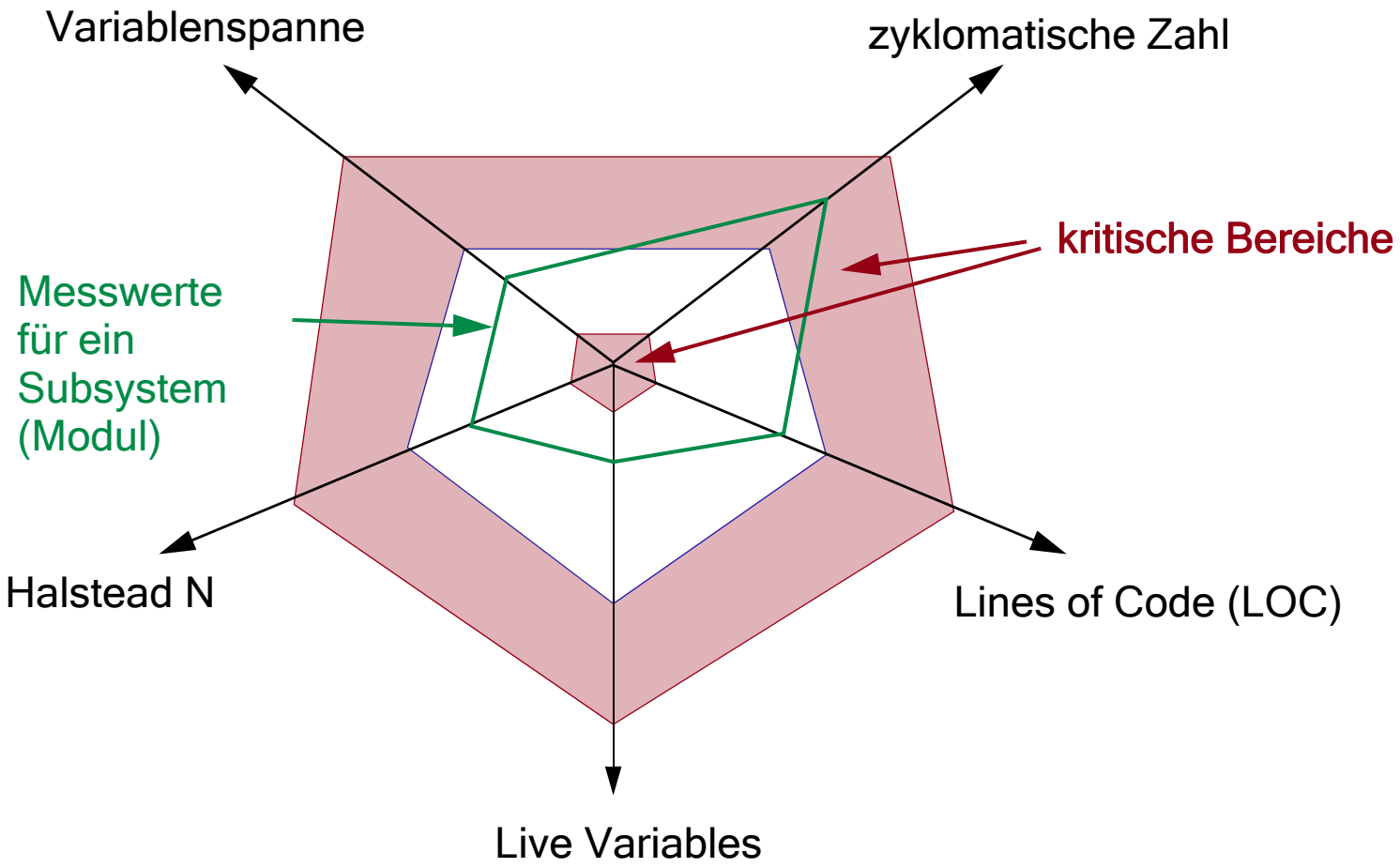
2. solche Grenzwerte für Messergebnisse sind nicht bekannt

   ⇨ alle Komponenten (Module, Klassen, Methoden, ...) werden untersucht, deren Messwerte ausserhalb des Bereichs liegen, in dem 95% der Messwerte liegen (oder 80% oder ...)

3. funktionaler Zusammenhang zwischen Metrik und gewünschtem Qualitätsmerkmal genauer bekannt

   ⇨ zulässige Werte für Metrik werden aus Qualitätsanforderungen errechnet (ein Wunschtraum ...)

Gleichzeitige Darstellung mehrere Messwerte mit Kiviatdiagramm:



Lines Of Code = LOC:

Die Zeilenzahl des Quelltextes ist die naheliegendste Metrik für ein Softwaresystem (betrachtetes Programmteil).

Doch wie legt man die „**Lines Of Code = LOC**“ einen Programmteil fest?

Möglichkeiten:

- Anzahl aller Zeilen der Textdatei(en) des betrachteten Programmteils
- Anzahl Zeilen der Textdatei(en) ohne Kommentare und Leerzeilen
- Anzahl Trennzeichen zwischen Anweisungen, also „;“ oder ...
- Anzahl Trennzeichen wie „;“ plus Schlüsselwörter wie „IF“, ...
- Anzahl Knoten im Kontrollflussgraphen (siehe folgende Folien)
- Programmlänge nach Halstead (siehe folgende Folien)

Lines of Code (LOC):

LOC(Programmteil) = Anzahl der Knoten im Kontrollflussgraphen dazu

Beispiel:

LOC(countVowels) = 7 (oder 8 ohne init-Segment)

Idee dieser Maßzahl:

- ⇨ betrachtete Programmteile oder ganze Programme mit hoher LOC sind zu komplex (no separation of concerns) und deshalb fehlerträchtig
- ⇨ [Programmteile mit geringer LOC sind zu klein und führen zu unnötigen Schnittstellenproblemen]

Probleme mit dieser Maßzahl:

- ⇨ Kanten = Kontrollflusslogik spielen keine Rolle
- ⇨ wie bewertet man geerbten Code einer Klasse

Zyklomatische Zahl nach McCabe:

ZZ(Programmteil) = |E| - |N| + 2k

mit G als Kontrollflussgraph des untersuchten Programmteils und

- ⇨ |E| := Anzahl Kanten von G
- ⇨ |N| := Anzahl Knoten von G
- ⇨ k := Anzahl Zusammenhangskomponenten von G
  (Anzahl der nicht miteinander verbundenen Teilgraphen von G)

Beispiel:

ZZ(countVowels) = 8 - 7 + 2 = 3

Regel von McCabe:

ZZ eines in sich abgeschlossenen Teilprogramms (Zusammenhangskomponente) sollte nicht höher als 10 sein, da sonst Programm zu komplex und zu schwer zu testen ist.

Interpretation und Probleme mit der zyklomatischen Zahl:

- es wird die Anzahl der Verzweigungen (unabhängigen Pfade) in einem Programm gemessen
  - ⇨ es wird davon ausgegangen, dass jede Zusammenhangskomponente (Teilprogramm) genau einen Eintritts- und einen Austrittsknoten hat
  - ⇨ damit besitzt jede Zusammenhangskomponente mit *n* Knoten mindestens *n-1* Kanten; diese immer vorhandenen Kanten werden nicht mitgezählt
  - ⇨ die kleinste Komplexität einer Zusammenhangskomponente soll *1* sein, also wird von der Anzahl der Kanten *n* abgezogen und *2* addiert
- in GOTO-freien Programmen wird damit genau die Anzahl der bedingten Anweisungen und Schleifen (if/while-Statements) gemessen
- die Zahl ändert sich nicht beim Einfügen normaler Anweisungen
- deshalb ist die Regel von McCabe mit **ZZ(Komponente) < 11** umstritten, da allenfalls eine Aussage über Testaufwand (Anzahl der zu testenden unabhängigen Programmpfade) getroffen wird

Halstead-Metriken - Eingangsgrößen:

Die Halstead-Metriken messen verschiedene Eigenschaften einer Software-komponente. Als Eingabe dienen immer:

- η_1 : Anzahl der unterschiedlichen Operatoren eines Programms (verwendete arithmetische Operatoren, Prozeduren, Methoden, ...)
- η_2: Anzahl der unterschiedlichen Operanden eines Programms (verwendete Variablen, Parameter, Konstanten, ...)
- $\mathbf{N_1}$: Gesamtzahl der verwendeten Operatoren in einem Programm (jede Verwendungsstelle wird separat gezählt)
- $\mathbf{N_2}$: Gesamtzahl der verwendeten Operanden in einem Programm (jede Verwendungsstelle wird separat gezählt)
- $\eta := \eta_1 + \eta_2$: Anzahl der verwendeten Deklarationen (**Programmvokabular**)
- $\mathbf{N} := N_1 + N_2$: Anzahl der angewandten Auftreten von Deklarationen (wird auch „normale“ **Programmlänge** genannt)

Halstead-Größen eines Programms (Prozedur, Methode):

```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);          (* start *)
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)
VAR i: INTEGER;

BEGIN

   count := 0; i := 0;          (* init *)

   WHILE s[i] # '.' DO

      IF    s[i]= 'a' OR s[i]= 'e' OR
            s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

            count := count+1;

      END;

      i := i+1;

END countVowels;          (* final *)
```

η_1 = Anzahl unterschiedlicher Operatoren =

η_2 = Anzahl unterschiedlicher Operanden =

$\mathbf{N_1}$ = Gesamtzahl verwendeter Operatoren =

$\mathbf{N_2}$ = Gesamtzahl verwendeter Operanden =

$\eta := \eta_1 + \eta_2 =$

$\mathbf{N} := N_1 + N_2 =$

Achtung: es ist Vereinbarungssache, ob Kontrollstrukturen, Klammen etc. wie „WHILE“, „IF“, „(“ auch als Operatoren gezählt werden.

In Literatur vorgeschlagene Zählregeln für Java-Programme:

- Arithmetische und logische Standardoperatoren:
  !, !=, %, %=, &, &&, &=, ...
- Auch weitere Operatoren (Sonderzeichen):
  - ⇨ = also Zuweisung
  - ⇨ ; Konkatenation von Anweisungen
  - ⇨ . Attributselektion
  - ⇨ (...) also Klammerungen in Ausdrücken
  - ⇨ ...
- Alle reservierten Java-Schlüsselwörter:
  extern, register, static, typedef, virtual, mutable, inline, break, enum, try, abstract, implements, extends, ...
- Definitionen von Methoden und Funktionen

Halstead-Metriken - Definition:

1. **Berechnete Programmlänge** $L := \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
   (hängt also nur von Anzahl verwendeter Operatoren und Operanden ab; postuliert wird, dass man mit einer festen Anzahl von Operatoren und Operanden immer Programme einer bestimmten logischen Größe schreibt)

2. **Programmgröße** $V = N \log_2 \eta$ (Programme Volume)
   (optimale Codierung des Programms als Bitvektor)

3. ...

Bewertung:

Es gibt eine ganze Reihe weiterer Halstead-Metriken, die versuchen zu bewerten:

- ⇨ **Schwierigkeit** der Erstellung eines Programms
- ⇨ **Adäquatheit** einer bestimmten Programmiersprache für Problemstellung
- ⇨ **Aufwand** für Erstellung eines Programms

Nutzen dieser Metriken ist stark umstritten!

„Live Variables“ und Variablenspanne:

- „**Live Variables**“ einer Komponente ist durchschnittliche Anzahl lebendiger Variablen in einem Programm pro Zeile
- eine Variable ist von ihrer ersten Definitionsstelle (vom Startknoten aus) bis zur letzten Definitions- oder Referenzierungsstelle (vor dem Endknoten) **lebendig**
- „**Variablenspanne“** einer Komponente ist die durchschnittliche Spanne zweier direkt aufeinander folgender definierender oder referenzierender Auftreten derselben Variable
- in beiden Fällen wird der Kontrollfluss des Programms (Schleifen, bedingte Verzweigungen) ignoriert und es werden einfach „nur“ Programmzeilen gezählt

Anmerkung:

Mit diesen beiden Metriken versucht man nicht die „Kontrollflusskomplexität“ oder einfache Größe einer Softwarekomponente, sondern die Komplexität des Datenflusses zu bewerten (wieviele Variablen muss man wie lange beim Erstellen von Programm-zeilen „im Kopf behalten“).

Beispiel für „Live Variables“ und „Variablenspanne“:

```
1:  IF min > max THEN
2:      hilf := min;
3:      min := max;
4:      max := hilf;
```

Berechnung der Metriken:

Zeile	define	reference	lebendige Variablen	Anzahl
1	---	min,max	min, max	2
2	hilf	min	min, max, hilf	3
3	min	max	min, max, hilf	3
4	max	hilf	max, hilf	2

Live Variables = 2,5

Variablenspanne = $((1+1)_{\text{für min}} + (2+1)_{\text{für max}} + (2)_{\text{für hilf}}) / 5 = 1{,}4$

Erste Überlegungen zu Metriken für objektorientierte Programme:

Betrachtet wird oft Kopplung von Klassen = Benutzt-Beziehungen zwischen Klassen:

- ⇨ **geringer fan-out** (wenige auslaufende Benutzt-Beziehungen) ist positiv, da sich dann eine Klasse auf wenige andere Klassen abstützt
- ⇨ **hoher fan-in** (viele einlaufende Benutzt-Beziehungen) ist positiv, da dann eine Klasse von vielen Klassen (wieder-)verwendet wird
- ⇨ beides kann nicht maximiert werden, da über alle Klassen hinweg gilt: **Summe fan-in = Summe fan-out**

Eine Klasse A benutzt eine Klasse B, wenn:

- ⇨ in A ein Verweis auf Objekt der Klasse B verwendet wird
- ⇨ in A eine Operation einen Parameter der Klasse B verwendet
- ⇨ in A eine Operation der Klasse B aufgerufen wird

Achtung: ähnlich kann man Kopplung von Modulen (Pakete) bzw. Kopplung von Methoden studieren.

Metriken für OO-Programme - weitere Überlegungen:

Gesucht werden Metriken, die neben der Kopplung von Klassen folgende Aspekte in Maßzahlen zusammenfassen:

- die Methoden einer Klasse sollten **enge Bindung (high cohesion)** besitzen, also einem ähnlichen Zweck dienen (wie misst man das?)
- die Klassen einer Vererbungshierarchie sollten ebenfalls **enge Bindung** besitzen
- die in einem Modul bzw. Paket zusammengefassten Klassen oder die in einer Klasse zusammengefassten Methoden sollten **enge Bindung** besitzen
- Klassen in verschiedenen Modulen sollte **lose gekoppelt** sein (wie misst man das?) **(loose coupling)**
- Klassen und Module bzw. Pakete sollten ein Implementierungsgeheimnis verbergen **(data abstraction, encapsulation)**
- ...

Bindungsmetriken - LOCOM1 (Low Cohesion Metric):

Die Bindung der Methoden einer Klasse wird untersucht. Methoden sind eng aneinander gebunden, wenn sie auf viele gemeinsame Attribute/Felder zugreifen:

- ⇨ P := Anzahl der Paare von Methoden ohne gemeinsame Attributzugriffe
- ⇨ Q := Anzahl der Paare von Methoden mit gemeinsamen Attributzugriffen
- ⇨ **LOCOM1** := if P > Q then P-Q else 0

Gewünscht wird Wert von LOCOM1 nahe bei 0.

Beispiel:

getName und setName der Klasse Person greift auf Attribut name zu
getAddr und setAddr der Klasse Person greift auf Attribut addr zu
compare greift auf Attribute name und addr zu

Es gilt dann:

es gibt 10 verschiedene Paarungen und P = 4 und Q = 6, also LOCOM1 = 0

Bindungsmetriken - LOCOM2 (Low Cohesion Metric):

Die Bindung der Methoden einer Klasse wird untersucht. Methoden sind eng aneinander gebunden, wenn sie auf viele gemeinsame Attribute/Felder zugreifen:

- m := Anzahl Methoden m_i einer Klasse
  $m(a_i)$:= Anzahl Methoden, die auf Attribut a_i zugreifen
- a := Anzahl Attribute a_i einer Klasse
- **LOCOM2** := $1 - (m(a_1) + ... + m(a_a)) / (m*a)$

Gewünscht wird kleiner Wert von LOCOM2 (z.B. kleiner 0,3 = 30%).

Beispiel:

getName und setName der Klasse Person greift auf Feld name zu
getAddr und setAddr der Klasse Person greift auf Feld addr zu
compare greift auf Felder name und addr zu

Es gilt dann:

m(name) = m(addr) = 3 und LOCOM2 = 1 - (3+3) / 10 = 0,4

Weitere Metriken - Kopplung von Klassen - 1:

- **Afferent Coupling (Ca/AC)**:
  Die Anzahl der Klassen ausserhalb eines betrachteten Teilsystems (Kategorie), die von den Klassen innerhalb des Teilsystems abhängen

- **Efferent Coupling (Ce/EC)**:
  Die Anzahl der Klassen innerhalb eines betrachteten Teilsystems (Kategorie), die von Klassen ausserhalb des betrachteten Teilsystems abhängen

- **Instabilität (I)**: I = Ce / (Ce+Ca)
  - ⇨ I hat einen Wert zwischen 0 und 1, falls nicht Ce + Ca = 0 gilt
    mit 0 = max. stabil u. 1 = max. unstabil
  - ⇨ der Wert 1 besagt, dass Ca = 0 ist; das betrachtete Teilsystem exportiert also nichts nach außen (keine Klassen und deren Methoden)
  - ⇨ der Wert 0 besagt, dass Ce = 0 ist; das betrachtete Teilsystem importiert also nichts von außen (keine Klassen und deren Methoden)
  - ⇨ Der „undefinierte“ Fall Ca = 0 und Ce = 0 kann nur auf ein (sinnloses) isoliertes Teilsystem zutreffen, das weder importiert noch exportiert

Weitere Metriken - Kopplung von Klassen - 2:

- **Coupling** als **Change Dependency between Classes (CDBC)**:
  CDBC zwischen Client Class CC und Server Class SC :=
  - ⇨ n falls SC Oberklasse von CC ist (n = Anzahl Methoden in CC)
  - ⇨ n falls CC ein Attribut des Typs SC hat
  - ⇨ j falls SC in j Methoden von CC benutzt wird
    (als Typ lokaler Variable, Parameter oder Methodenaufruf von SC)

  CDBC bewertet den Aufwand, der mit Änderung der Klasse SC in CC verbunden sein könnte (sollte möglichst gering sein).

- **Encapsulation** als **Attribute Hiding Factor (AHF)**:
  - ⇨ Summe der Unsichtbarkeiten aller Attribute in allen Klassen geteilt durch die Anzahl aller Attribute
  - ⇨ Unsichtbarkeit eines Attributs := Prozentzahl der Klassen, für die das Attribut nicht sichtbar ist (abgesehen von eigener Klasse)

  Sind alle Attribute als „private“ definiert, dann ist AHF = 1.

Weitere Metriken:

- **Tiefe von Vererbungshierarchien**:
  zu tiefe Hierarchien werden unübersichtlich; man weiss nicht mehr, was man erbt
- **Breite von Vererbungshierarchien**:
  zu breite Vererbungshierarchien deuten auf Fehlen von zusammenfassenden Klassen hin
- **Anzahl Redefinitionen in einer Klassenhierarchie**:
  je mehr desto gefährlicher
- **Anzahl Zugriffe auf geerbte Attribute**:
  sind ebenfalls gefährlich, da beim Ändern von Attributen oder Attributzugriffen in Oberklasse die Zugriffen in den Unterklassen oft vergessen werden
- **Halstead-Metriken**: ...
- ...

Komplexitätsmaße:

- **Response For Class (RFC)**:
  die Anzahl der in der Klasse deklarierten Methoden + die Anzahl der geerbten Methoden + die Anzahl sichtbarer Methoden anderer Klassen
  (Alle Methoden, die aufgerufen werden können? Sehr schwammig definiert!!!)

- **Weighted Methods Per Class (WMPC1)**:
  die Summe der zyklomatischen Zahlen ZZ aller Methoden der Klasse (ohne geerbte Methoden)

- **Number of Remote Methods (NORM)**:
  die Anzahl der in einer Klasse gerufenen Methoden „fremder“ Klassen (also nicht die Klasse selbst oder eine ihrer Oberklassen)

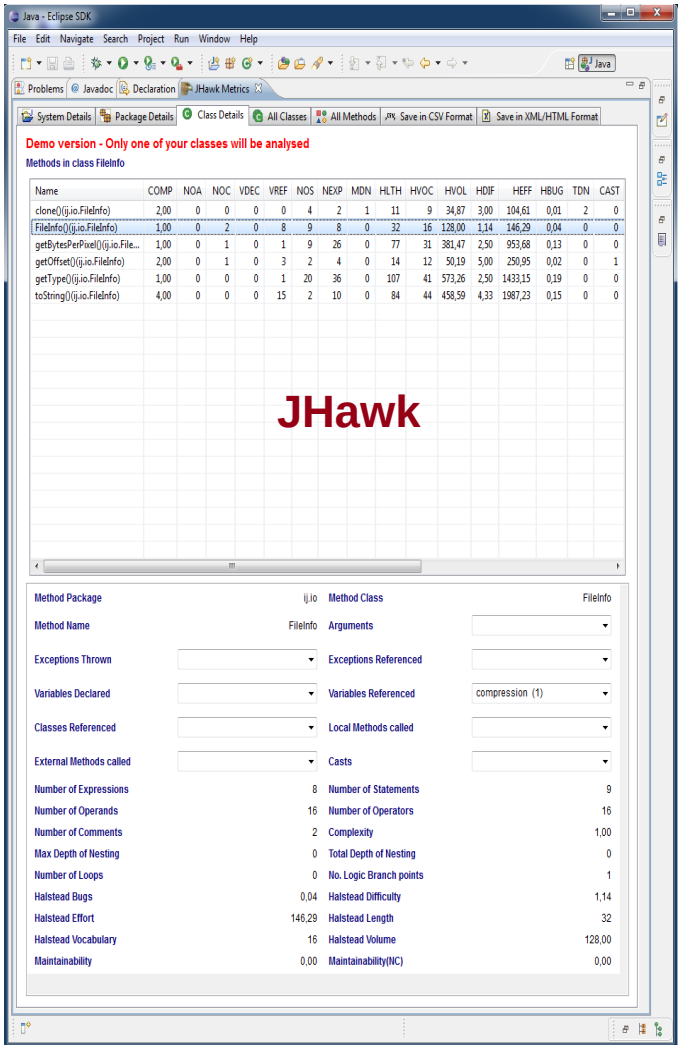
- **Attribute Complexity (AC)**:
  die gewichtete Summe der Attribute einer Klasse wird gebildet; Gewichte werden gemäß Typen/Klassen der Attribute vergeben.

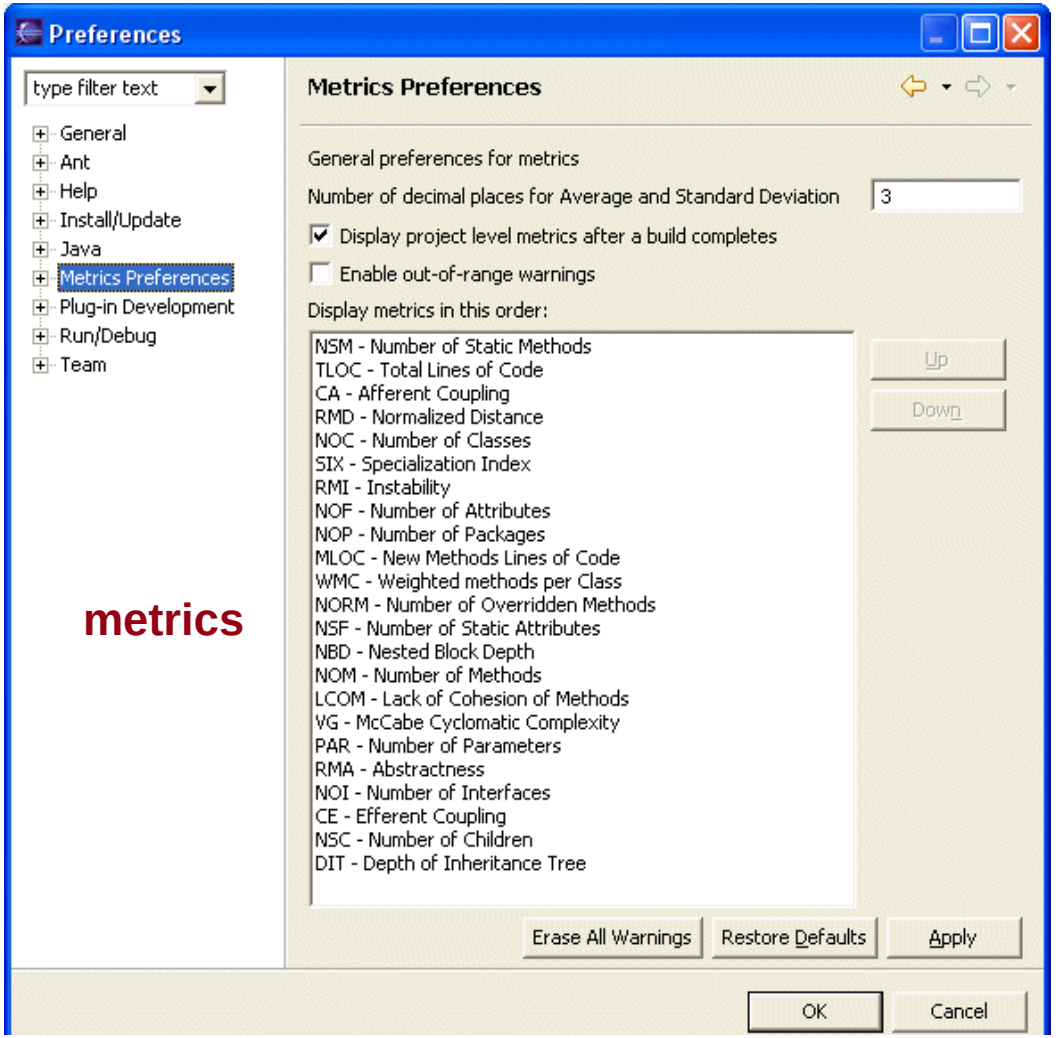
- ...

Beispiele für „Mess“-Werkzeuge:

- JHawk (http://www.virtualmachinery.com/jhawkprod.htm) für Java:
  - ⇨ unterstützt viele Zähl-Metriken (Anzahl von Parametern, ...)
  - ⇨ unterstützt die klassischen zweifelhaften Metriken (Halstead, ...)
  - ⇨ kommerzielles Produkt
- metrics (http://metrics.sourceforge.net/) für Java:
  - ⇨ „Open Source“-Software
  - ⇨ unterstützt deutlich weniger Metriken als JHawk
  - ⇨ aber ebenfalls Ca, Ce, Tiefe von Vererbungshierarchien, ...
- Together (scheint von Borland nicht mehr unterstützt zu werden)
  - ⇨ UML-Werkzeug mit enger Modell-Code-Integration
  - ⇨ unterstützt(e) viele Metriken für C++ und Java
  - ⇨ fast alle hier vorgestellten komplexeren OO-Metriken stammen aus der Together-Dokumentation

Screendumps von JHawk und metrics:


Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Problems Javadoc Declaration JHawk Metrics
System Details Package Details Class Details All Classes All Methods Save in CSV Format Save in XML/HTML Format
Demo version - Only one of your classes will be analysed
Methods in class FileInfo
Name COMP NOA NOC VDEC VREF NOS NEXP MDN HLTH HVOC HVOL HDIF HEFF HBUG TDN CAST
clone()(ij.io.FileInfo) 2,00 0 0 0 0 4 2 1 11 9 34,87 3,00 104,61 0,01 2 0
FileInfo()(ij.io.FileInfo) 1,00 0 2 0 8 9 8 0 32 16 128,00 1,14 146,29 0,04 0 0
getBytesPerPixel()(ij.io.File... 1,00 0 1 0 1 9 26 0 77 31 381,47 2,50 953,68 0,13 0 0
getOffset()(ij.io.FileInfo) 2,00 0 1 0 3 2 4 0 14 12 50,19 5,00 250,95 0,02 0 1
getType()(ij.io.FileInfo) 1,00 0 0 0 1 20 36 0 107 41 573,26 2,50 1433,15 0,19 0 0
toString()(ij.io.FileInfo) 4,00 0 0 0 15 2 10 0 84 44 458,59 4,33 1987,23 0,15 0 0
JHawk
Method Package ij.io
Method Class FileInfo
Method Name FileInfo
Arguments
Exceptions Thrown
Exceptions Referenced
Variables Declared
Variables Referenced compression (1)
Classes Referenced
Local Methods called
External Methods called
Casts
Number of Expressions 8
Number of Statements 9
Number of Operands 16
Number of Operators 16
Number of Comments 2
Complexity 1,00
Max Depth of Nesting 0
Total Depth of Nesting 0
Number of Loops 0
No. Logic Branch points 1
Halstead Bugs 0,04
Halstead Difficulty 1,14
Halstead Effort 146,29
Halstead Length 32
Halstead Vocabulary 16
Halstead Volume 128,00
Maintainability 0,00
Maintainability(NC) 0,00



3.6 Zusammenfassung

Die **Visualisierung von Software** ist sowohl beim „Forward Engineering“ für den Entwurf neuer Programmarchitekturen als auch beim „Reverse Engineering“ für das Studium von „Legacy Software“ mit unbekannter Programmstruktur sehr hilfreich.

Werkzeugunterstützte **statische Analyseverfahren** helfen frühzeitig bei der Identifikation kritischer Programmstellen. Es sollten folgende Analyseverfahren immer eingesetzt werden:

- ⇨ **Stilanalyse** (Überprüfung vereinbarter Programmierkonventionen)
- ⇨ **„dead code“-Analyse** (oft in Compiler eingebaut): nie verwendete Methoden, Variablen, Parameter, ... (wurde bisher nicht angesprochen)
- ⇨ **Datenflussanalyse** (wenn Werkzeug verfügbar)

Weitere Analyseverfahren und vor allem **Metriken** sollten in großen Projekten zumindest versuchsweise eingesetzt werden.

Vorgehensweise beim Einsatz von Maßen aus [Li02]:

1. **Fragen zur Ausgangssituation**:
   - ⇨ In welcher Phase (Aktivitätsbereich) des Softwareentwicklungsprozesses soll eine Verbesserung eingeführt werden (z.B. Design, Codierung, ...)?
   - ⇨ Was soll damit erreicht werden bzw. welche Art von Fehler soll reduziert werden (z.B. Reduktion C++ Codierungsfehler)?
   - ⇨ Welche Methode soll eingesetzt werden (z.B. OO-Metriken)?
   - ⇨ Welche Technik/Werkzeug soll eingesetzt werden
2. **Bewertung des aktuellen Standes** des Entwicklungsprozesses:
   - ⇨ Welche Kosten u. welcher Aufwand entstehen in welcher Phase?
   - ⇨ Wie ist die Qualität der Ergebnisse jeder Phase?
   - ⇨ In welcher Phase entsteht welcher Anteil an Fehlern und welcher Teil der Fehlerbeseitigungskosten?

Vorgehensweise - Fortsetzung:

3. Mittel zur Bestimmung des aktuellen Standes, **zu messende Aspekte**:
   - ⇨ Kosten- und Zeitverfolgung beim Entwicklungsprozess
   - ⇨ Definition von Qualitätsmaßen für Produkt pro Phase
   - ⇨ Erhebung von Fehlerstatistiken
4. **Analyse der Ergebnisse** und Erarbeitung von Verbesserungsvorschlägen:
   - ⇨ Auswertung der Maße
   - ⇨ Definition von Zielen auf Basis der Messwerte
   - ⇨ Entscheidung für Verbesserung in bestimmten Phasen
   - ⇨ Auswahl geeigneter Methoden und Werkzeuge
   - ⇨ Einführung der Methoden und Werkzeuge in Entwicklungsprozess
5. **Bewertung** der durchgeführten Änderungen:
   - ⇨ Kontinuierliche Weiterauswertung der Maße
   - ⇨ erneute Analyse nach „Abklingen von Einschwingvorgängen“

Abstraktes Fallbeispiel aus [Li02]:

1. Angestrebt wird die deutliche Reduktion der Fehleranzahl mit dem Ziel den Kunden in Zukunft zuverlässigere Produkte zur Verfügung zu stellen
2. Minimiert wird dafür das **Zuverlässigkeitsmaß MTBF** = „Mean Time Between Failure“ (mittlere verstrichene Zeit zwischen zwei Fehlern):
   - ⇨ innerhalb von 343 Tagen wurden 17 (schwerwiegende) Fehler gemeldet
   - ⇨ MTBF = 20,2 - alle 20,2 Tage tritt durchschnittlich ein Fehler auf
3. Die gemeldeten Fehler lassen sich wie folgt den Phasen des Entwicklungs-prozesses zuordnen und der Aufwand für ihre Behebung ist wie folgt
   - ⇨ 5 Fehler aus der Anforderungsdefinitionsphase mit 27 PT durchschnittlichem Korrekturaufwand (PT = Personentage)
   - ⇨ 3 Fehler aus der Entwurfsphase mit 5,7 PT Korrekturaufwand
   - ⇨ 10 Fehler aus der Implementierungsphase mit 0,6 PT Korrekturaufwand

Abstraktes Fallbeispiel - Fortsetzung :

4. Ziele zur Verbesserung des Entwicklungsprozesses und Softwareprodukts:
   - ⇨ Reduktion der Fehleranzahl in der Definitionsphase zur deutlichen Aufwandsreduktion
   - ⇨ Reduktion der Fehleranzahl in der Implementierungsphase zur deutlichen Verbesserung der Qualität der ausgelieferten Software

5. Auswahl von Verbesserungsmaßnahmen in der Anforderungsdefinitionsphase:
   - ⇨ falls ungenaue Spezifikation der Anforderungen des Kunden, dann Einsatz von (semi-)formalen Spezifikationstechniken
   - ⇨ falls genaue Spezifikation falscher Anforderungen des Kunden, dann Einsatz von Rapid-Prototyping-Vorgehensweise

6. Auswahl von Verbesserungsmaßnahmen in der Implementierungsphase
   - ⇨ hätte Datenflussanalyse die Fehler gefunden
   - ⇨ hätte Metrix X fehlerhaften Code als „qualitativ zweifelhaft“ erkannt
   - ⇨ hätten systematischere Testverfahren die Fehler aufgedeckt

3.7 Zusätzliche Literatur

[Ka00] St. Kan: Metrics and Models in Software Quality Engineering, Addison-Wesley, 2nd Ed. (2003), 528 Seiten

[SD98] J. Stasko, J. Domingue, M. Brown et al.: *Software Visualization*, MIT Press (1998), 562 Seiten

[SG96] M. Shaw, D. Garlan: *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, (1996)

[Tha00] G. Thaller: *Software-Metriken einsetzen, bewerten, messen*, VT Verlag Technik Berlin, 2. Auflage (2000), 208 Seiten

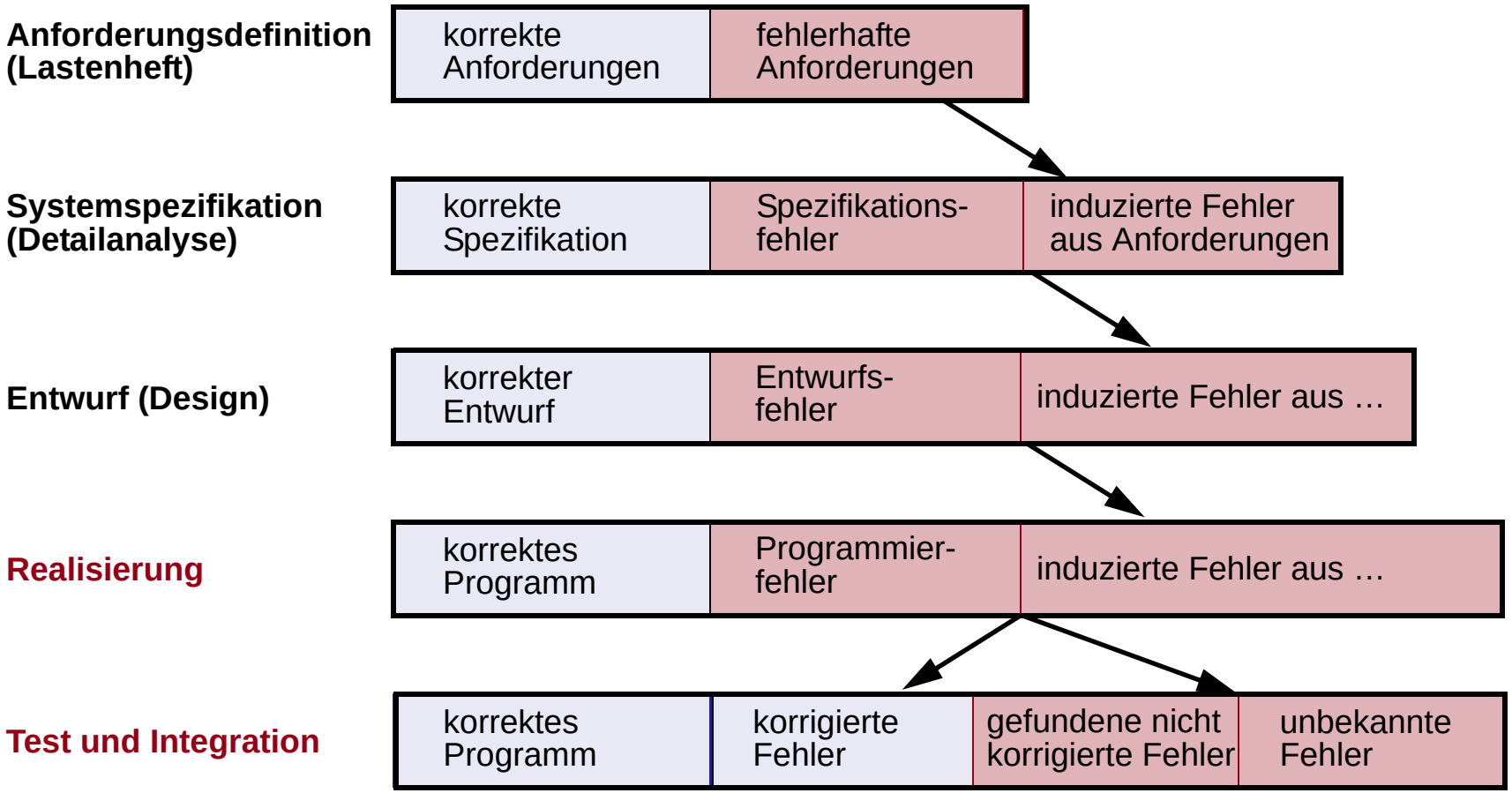
4. Dynamische Programmanalysen und Testen

The older I get, the more aggressive I get about testing. I like Kent Beck's rule of thumb that a developer should write at least as much test code as production code. Testing should be a continuous process. No code should be written until you know how to test it. Once you have written it, write the tests for it. Until the test works, you cannot claim to have finished writing the code. [FS98]

Lernziele:

- Schwierigkeiten des systematischen Tests von Software verstehen
- Minimalstandards für Test von Software kennenlernen
- wichtige Werkzeuge für Performanzanalyse kennenlernen
- Werkzeuge für Identifikation von Speicherlecks, ... kennenlernen

4.1 Einleitung



Fehlerzustand, Fehlerwirkung und Fehlhandlung (DIN 66271):

- **Fehlerzustand (fault):**
  - ⇨ inkorrektes Teilprogramm, inkorrekte Anweisung oder Datendefinition, die Ursache für Fehlerwirkung ist
  - ⇨ Zustand eines Softwareprodukts oder einer seiner Komponenten, der unter spezifischen Bedingungen eine geforderte Funktion beeinträchtigen kann
- **Fehlerwirkung (failure):**
  - ⇨ Wirkung eines Fehlerzustandes, die bei der Ausführung des Testobjektes nach „aussen“ in Erscheinung tritt
  - ⇨ Abweichung zwischen spezifiziertem Soll-Wert (Anforderungsdefinition) und beobachtetem Ist-Wert (bzw. Soll- und Ist-Verhalten)
- **Fehlhandlung (error):**
  - ⇨ menschliche Handlung (des Entwicklers), die zu einem Fehlerzustand in der Software führt
  - ⇨ menschliche Handlung (des Anwenders), die ein unerwünschtes Ergebnis zur Folge hat

Ursachenkette für Fehler (in Anlehnung an DIN 66271):

- jeder Fehler (**fault**) oder Mangel ist seit dem Zeitpunkt der Entwicklung in der Software vorhanden - Software nützt sich nicht ab
- er ist aufgrund des fehlerhaften Verhaltens (**error**) eines Entwicklers entstanden (und wegen mangelhafter Qualitätssicherungsmaßnahmen nicht entdeckt worden)
- ein Softwarefehler kommt nur bei der Ausführung der Software als Fehlerwirkung (**failure**) zum Tragen und führt dann zu einer ggf. sichtbaren Abweichung des tatsächlichen Programmverhaltens vom gewünschten Programmverhalten
- Fehler in einem Programm können durch andere Fehler **maskiert** werden und kommen somit ggf. nie zum Tragen (bis diese anderen Fehler behoben sind)

Validation und Verifikation (durch dynamische Tests):

- **Validation von Software:**

  Prüfung, ob die Software das vom Anwender „wirklich“ gewünschte Verhalten zeigt (in einem bestimmten Anwendungsszenario)

  ☞ Haben wir das richtige Softwaresystem realisiert?

- **Verifikation von Software:**

  Prüfung, ob die Implementierung der Software die Anforderungen erfüllt, die vorab (vertraglich) festgelegt wurden

  ☞ Haben wir das Softwaresystem richtig realisiert?

Achtung:

Eine „richtig realisierte“ = korrekte Software (erfüllt die spezifizierten Anforderungen) muss noch lange nicht das „wirklich“ gewünschte Verhalten zeigen!

Typische Programmierfehler nach [BP84]:

- **Berechnungsfehler**: Komponente berechnet falsche Funktion
  - ⇨ z.B. Konvertierungsfehler in Fortran bei Variablen, die mit I, J oder K anfangen und damit implizit als Integer deklariert sind
- **Schnittstellenfehler**: Inkonsistenz (bezüglich erwarteter Funktionsweise) zwischen Aufrufsstelle und Deklaration - siehe vor allem Abschnitt 4.3
  - ⇨ Übergabe falscher Parameter, Vertauschen von Parametern
  - ⇨ Verletzung der Randbedingungen, unter denen aufgerufene Komponente funktioniert
- **Kontrollflussfehler**: Ausführung eines falschen Programmpfades - siehe vor allem Abschnitt 4.4
  - ⇨ Vertauschung von Anweisungen
  - ⇨ falsche Kontrollbedingung (z.B. „kleiner“ statt „kleiner gleich“), “off by one”: Schleife wird einmal zuwenig oder zu oft durchlaufen

- **Datenflussfehler**: falscher Zugriff auf Variablen und Datenstrukturen - siehe vor allem Abschnitt 4.2 und Abschnitt 4.5
  - ⇨ Variable wird nicht initialisiert (Initialisierungsfehler)
  - ⇨ falsche Arrayindizierung
  - ⇨ Zuweisung an falsche Variable
  - ⇨ Zugriff auf Nil-Pointer oder bereits freigegebenes Objekt
  - ⇨ Objekt wird nicht freigegeben
- **Zeitfehler**: gefordertes Zeitverhalten wird nicht eingehalten - siehe Abschnitt 4.2
  - ⇨ Implementierung ist nicht effizient genug
  - ⇨ wichtige Interrupts werden zu lange blockiert
- **Redefinitionsfehler**: geerbte Operation wird nicht semantikerhaltend redefiniert - siehe Abschnitt 4.6
  - ⇨ ein „Nutzer“ der Oberklasse geht von Eigenschaften der aufgerufenen Operation aus, die Redefinition in Unterklasse nicht (mehr) erfüllt

Beispiel für fehlerhafte Prozedur:

```
PROCEDURE countVowels(s: Sentence; VAR count: INTEGER);
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN
   count := 0;                                          (* Initialisierungsfehler - i wird nicht initialisiert. *)

   WHILE s[i] # '.' DO

      IF s[i]= 'a' OR s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'   (* Kontrollflussfehler - keine Prüfung auf 'e'. *)
      THEN
         count := count + 2;                            (* Berechnungsfehler - count wird um 2 erhöht. *)
      END;
      count := i+1;                                     (* Datenflussfehler - Zuweisung nicht an i. *)

   END (* WHILE *)

END countVowels

   ...

countVowels('to be . . . or not to be.', count);        (*Schnittstellenfehler - dot im Satz. *)
```

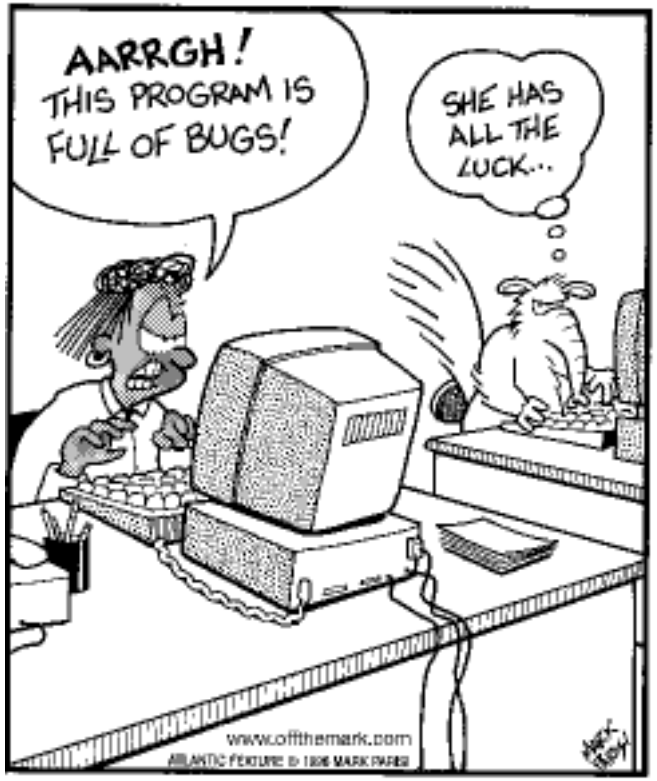
Was wird also getestet:

Testverfahren für Softwarekomponenten (Operation, Klasse, Modul/Paket, System) können danach klassifiziert werden, was getestet wird:

- **Funktionalitätstest**: das Ein-/Ausgabeverhalten der Software; das steht beim Testen (zunächst) stark im Vordergrund und wird in Abschnitt 4.3 bis Abschnitt 4.5 behandelt
- **[Benutzbarkeitstest**: es geht um die „gute“ Gestaltung der Benutzeroberfläche; schwieriges Thema, das hier nicht weiter vertieft wird**]**
- **Performanztest**: Laufzeitverhalten und Speicherplatzverbrauch einer Komponente werden gemessen und dabei oft durchlaufene ineffiziente Programmteile oder Speicherlecks identifiziert; siehe Abschnitt 4.2
- **Stresstest**: die Komponente wird mit sehr großen Datenmengen oder sehr großer Anzahl gleichzeitiger Anfragen oder ... getestet (siehe Auswahl von Testdaten in Abschnitt 4.3)

Wie wird getestet:

- **Funktionstest** (black box test): die interne Struktur der Komponente wird nicht betrachtet; getestet wird Ein-/Ausgabeverhalten gegen Spezifikation; siehe Abschnitt 4.3
- **Strukturtest** (white box test): interne Struktur der Komponente wird zur Testplanung und -Überwachung herangezogen:
  - ⇨ Kontrollflussgraph: siehe Abschnitt 4.4
  - ⇨ Datenflussgraph: siehe Abschnitt 4.5
  - ⇨ [Automaten: siehe Abschnitt 4.6]
- **Diversifikationstest**: Verhalten einer Komponentenversion wird mit Verhalten anderer Komponentenversionen verglichen



Diversifikationstestverfahren:

Das Verhalten verschiedener Varianten eines Programms wird verglichen.

- **Mutationstestverfahren**: ein Programm wird absichtlich durch Transformationen verändert und damit in aller Regel mit Fehlern versehen
  - ⇨ Einbau von Fehlern lässt sich (mehr oder weniger) automatisieren
  - ⇨ eingebaute Fehler entsprechen oft nicht tatsächlich gemachten Fehlern
  - ⇨ eingebaute Fehler stören Suche nach echten Fehlern
- **N-Versionen-Programmierung**: verschiedene Versionen eines Programms werden völlig unabhängig voneinander entwickelt
  - ⇨ sehr aufwändig, da man mehrere Entwicklerteams braucht (und gegebenfalls sogar Hardware mehrfach beschaffen muß)
  - ⇨ Fehler der verschiedenen Teams nicht unabhängig voneinander (z.B. haben alle Versionen dieselben Anforderungsdefinitionsfehlern)

Mutationstestverfahren:

Erzeugung von Mutationen durch:

Vertauschen von Anweisungsfolgen, Umdrehen von Kontrollflussbedingungen, Löschen von Zuweisungen, Ändern von Konstanten, …

Zielsetzungen:

- **Identifikation fehlender Testfälle**: für jeden Mutanten sollte mindestens ein Testfall existieren, der Original und Mutant unterscheiden kann
- **Eliminination nutzloser Testfälle**: Gruppe von Testfällen verhält sich bezüglich der Erkennung von Mutanten völlig gleich
- **Schätzung der Restfehlermenge** RF: RF ~ GF * (M/GM -1)

  mit GM = Anzahl gefundener eingebauter Fehler (Mutationen)
  M = Gesamtzahl absichtlich eingebauter Fehler
  GF = Anzahl gefundener echter Fehler
  [F = Gesamtzahl echter Fehler = GF + RF]

  Annahme: GF/F ~ GM/M (Korrelation gilt allenfalls für bestimmte Fehler)

N-Versionen-Programmierung (Back-to-Back-Testing):

Zielsetzungen:

- ⇨ eine Version kann als Orakel für die Korrektheit der Ausgaben einer anderen Version herangezogen werden (geht ab 2 Versionen)
- ⇨ Robustheit der ausgelieferten Software kann erhöht werden durch gleichzeitige Berechnung eines benötigten Ergebnisses durch mehrere Versionen einer Software
- ⇨ Liefern verschiedene Versionen unterschiedliche Ergebnisse, so wird Mehrheitsentscheidung verwendet (geht ab 3 Versionen)

Probleme:

- ⇨ N Versionen enthalten mehr Fehler als eine Version: falsche Versionen können richtige überstimmen oder gemeinsame Ressourcen blockieren
- ⇨ Fehler verschiedener Versionen sind nicht immer unabhängig voneinander: Fehler aus der Anforderungsdefinition oder typische Programmiersprachenfehler oder falsche Algorithmen können alle Versionen enthalten

Wann erhöhte Robustheit bei n-Versionen-Programmierung:

n Versionen enthalten in etwa *n*-mal so viele Fehler wie eine Version, aber falls Wahrscheinlichkeit *A* für fehlerhafte Ausgabe oder Ausfall einer Version *v* unabhängig vom Ausfall anderer Versionen ist, dann gilt:

- richtiges Ergebnis wird berechnet, solange höchstens (n - 1)/2 Versionen fehlerhaft arbeiten (*n* ist sinnvoller Weise ungerade Zahl)
- Wahrscheinlichkeit für gleichzeitigen Ausfall von *k* unabhängigen Versionen:

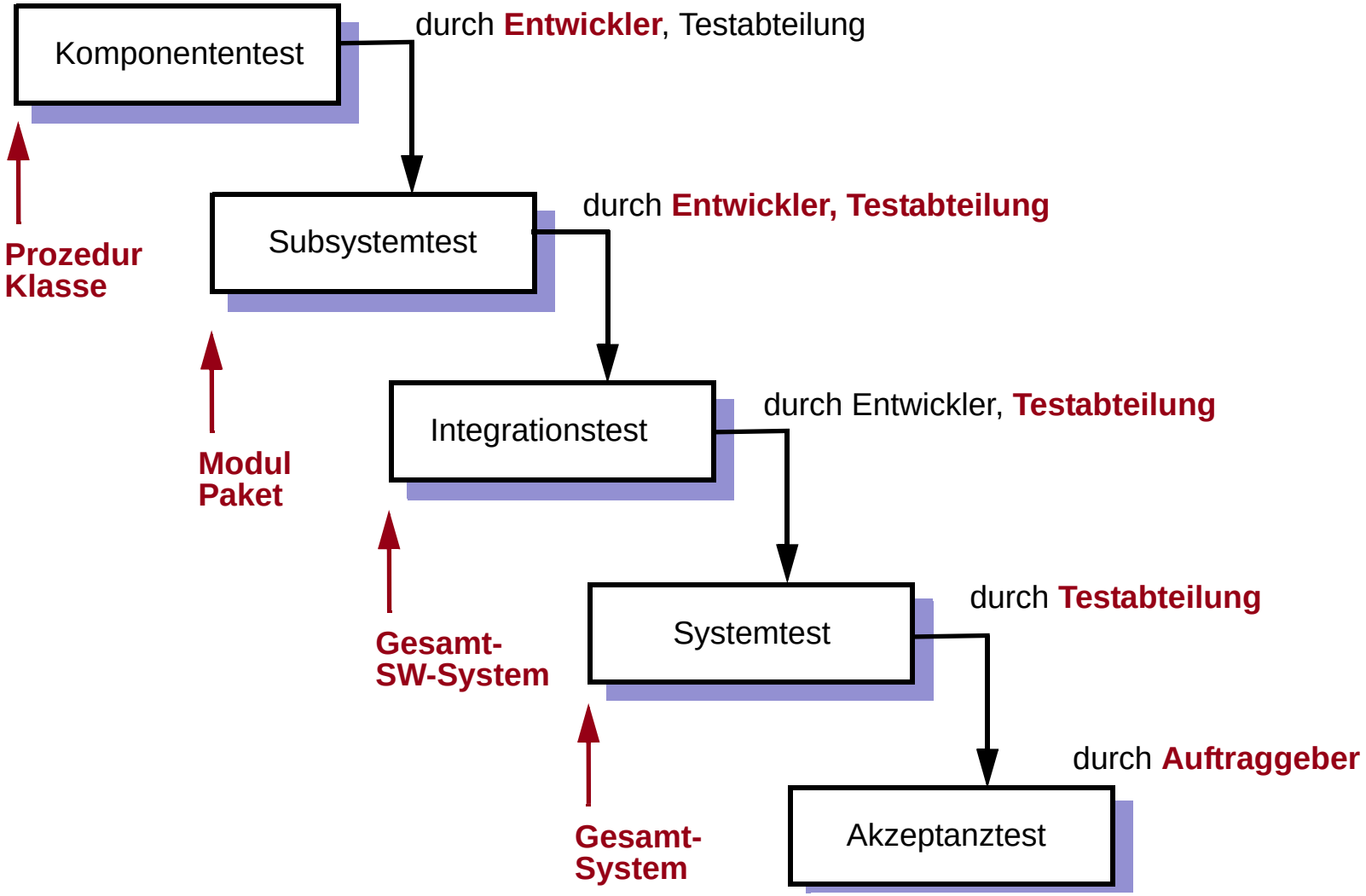
$$\binom{n}{k} \times A^k \times (1-A)^{(n-k)} = \frac{\prod_{i=n-k+1}^{n} i}{\prod_{i=1}^{k} i} \times A^k \times (1-A)^{(n-k}$$

- Wahrscheinlichkeit für **Ausfall des Gesamtsystems** mit *n* Versionen:

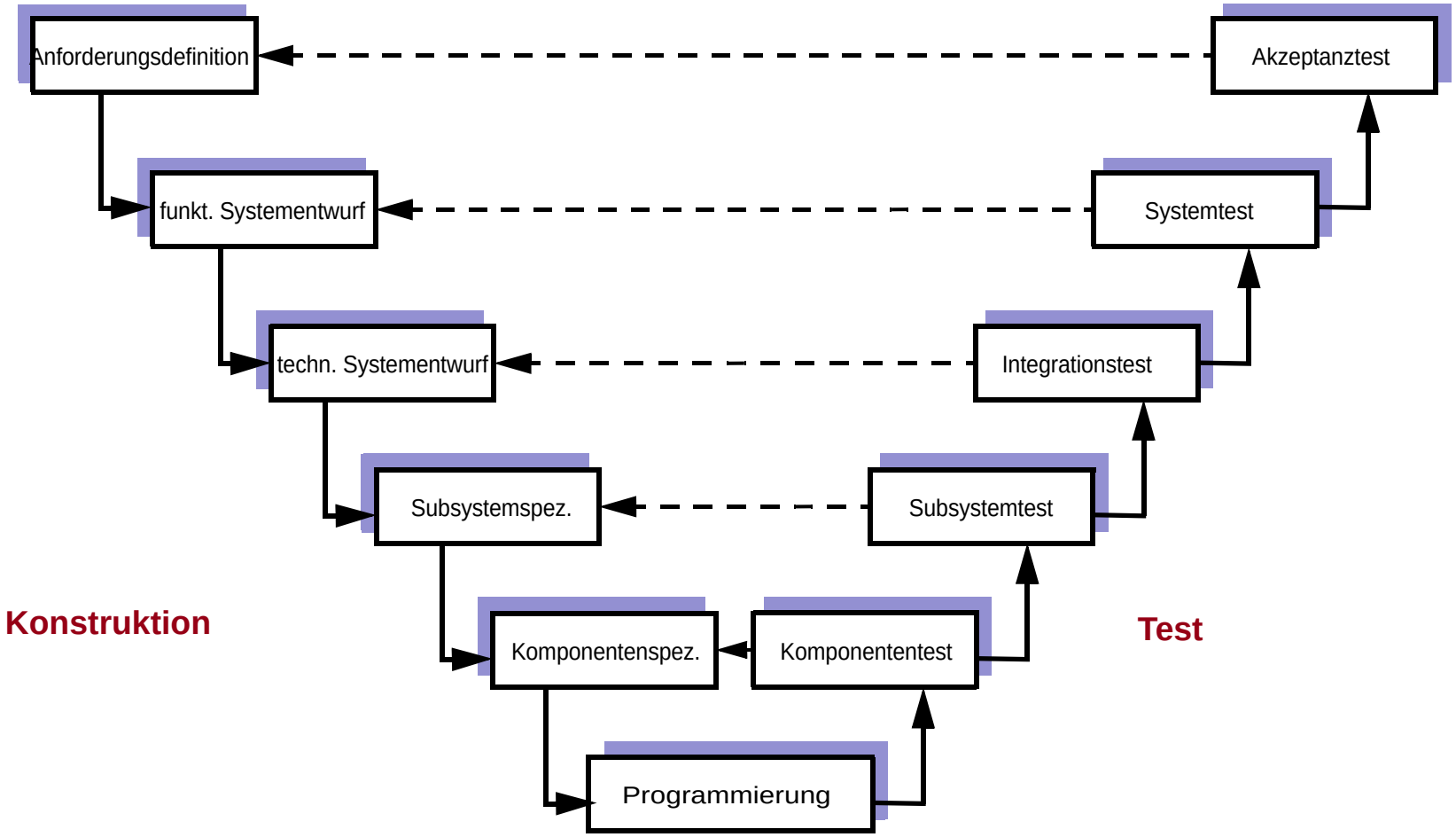
$$\sum_{k=(n+1)/2}^{n} \binom{n}{k} \times A^k \times (1-A)^{(n-k)}$$

(bei n = 3: Wahrscheinlichkeit für Ausfall von 2 oder 3 Versionen)

Der Testprozess als Bestandteil des Softwareentwicklungsprozesses:



Einbettung in modifiziertes V-Modell:



Komponententest (Unit-Test) und Subsystemtest:

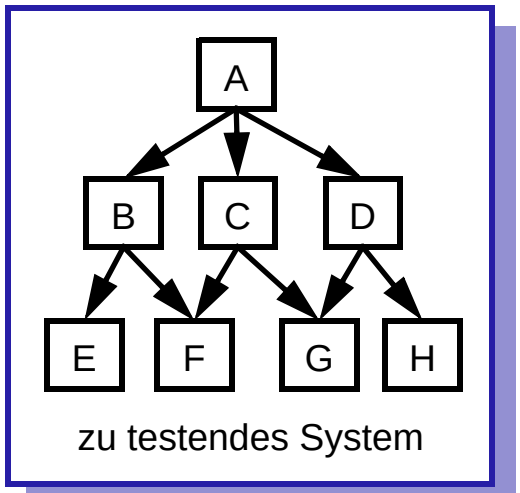
- jeweils ein **einzelner Softwarebaustein** wird überprüft, isoliert von anderen Softwarebausteinen des Systems
- die betrachtete Komponente (Unit) kann eine Klasse, Paket (Modul) sein
- **Subsystemtest** kann als Test einer besonders großen Komponente aufgefasst werden
- getestet wird gegen die Spezifikation der Schnittstelle der Komponente, dabei betrachtet werden funktionales Verhalten, Robustheit, Effizienz, ...
- Testziele sind das Aufdecken von Berechnungsfehlern, Kontrollflussfehlern, ...
- getestet wird in jedem Fall die Komponente für sich mit
  - ⇨ **Teststummel** (Platzhalter, Dummies, Stubs) für benötigte Dienste anderer Komponenten
  - ⇨ **Testtreibern** (Driver) für den (automatisierten) Test der Schnittstelle (für Eingabe von Parametern, Ausgabe von Parametern, ...)

Integrationstest:

- das gesamte Software-System (oder ein abgeschlossenes Teilsystem) wird getestet; Schwerpunkt liegt dabei auf Test des **Zusammenspiels** der Einzelkomponenten
- normalerweise wird vorausgesetzt, dass Einzelkomponenten vorab bereits getestet wurden
- auch hier müssen wieder Testtreiber geschrieben werden, die interaktive Benutzer-schnittstellen ersetzen und Tests automatisieren
- auf Teststummel kann meist verzichtet werden, da alle benötigten Teilsysteme zusammen getestet werden
- **Testziel** ist vor allem das Aufdecken von **Schnittstellenfehlern** und insbesondere Fehler beim Austausch von Daten

Gängige Integrationsteststrategien:

- **“Big Bang”-Strategie**: alle Teile sofort integrieren und nur als Gesamtheit testen
  - ⇨ Lokalisierung von Fehlern schwierig
  - ⇨ Arbeitsteilung kaum möglich
  - ⇨ Testen beginnt zu spät



- **“Top-down”-Testverfahren**: zuerst A mit Dummies für B, C und D; dann B mit Dummies für E und F, …
  - ⇨ Erstellung „vernünftiger“ Dummies schwierig
  - ⇨ Test der Basisschicht sehr spät
- **“Bottom-Up”-Testverfahren**: zuerst E, F, G und H mit Testtreibern, die Einbindung in B, C und D simulieren; dann B, C und D mit Testtreiber …
  - ⇨ Test des Gesamtverhaltens des Systems gegen Lastenheft erst am Ende
  - ⇨ Designfehler und Effizienzprobleme werden oft erst spät entdeckt

Gängige Integrationsteststrategien - Fortsetzung:

- **Inkrementelles Testen**: Grundgerüst vorhanden, weitere Komponenten werden stückweise hinzugefügt
  - ⇨ wie erstellt und testet man Grundgerüst (z.B. Top-Down-Testen mit Depth-First-Strategie)
  - ⇨ Hinzufügen von Komponenten kann bisherige Testergebnisse entwerten
- **Regressionstest**: Systemänderungen (bei Wartung oder inkrementellem Testen) können Fehler in bereits getesteten Funktionen verursachen
  - ⇨ möglichst viele Tests werden automatisiert
  - ⇨ bei jeder Änderung werden alle vorhandenen Tests durchgeführt
  - ⇨ neue Testergebnisse werden mit alten Testergebnissen verglichen

Achtung:

Nur inkrementelles Testen kombiniert mit Regressionstest passt zu den heute üblichen inkrementellen und iterativen Softwareentwicklungsprozessen (siehe Kapitel 5).

Systemtest - grundsätzliche Vorgehensweise:

- Nach abgeschlossenem Integrationstest und vor dem Abnahmetest erfolgt der Systemtest beim Softwareentwickler durch Kunden (**α-Test**)
- Variante: Systemtests bei ausgewählten Pilotkunden vor Ort (**β-Test**)
- Systemtest überprüft aus der **Sicht des Kunden**, ob das Gesamtprodukt die an es gestellten Anforderungen erfüllt (nicht mehr aus Sicht des Entwicklers)
- anstelle von Testtreibern und Teststummeln kommen nun soweit möglich immer die realen (Hardware-)Komponenten zum Einsatz
- Systemtest sollte nicht beim Kunden in der Produktionsumgebung stattfinden, sondern in möglichst realitätsnaher Testumgebung durchgeführt werden
- beim Test sollten die **tatsächlichen Geschäftsprozesse** beim Kunden berücksichtigt werden, in die das getestete System eingebettet wird
- dabei durchgeführt werden: Volumentests (große Datenmengen), Stresstests (Überlastung), Test auf Sicherheit, Stabilität, Robustheit, ...

Systemtest - nichtfunktionale Anforderungen:

- **Lasttest**: Messung des Systemverhaltens bei steigender Systemlast
- **Performanztest**: Messung der Verarbeitungsgeschwindigkeit unter bestimmten Randbedingungen
- **Kompatibilität**: Verträglichkeit mit vorhandenen anderen Systemen, korrekter Im- und Export externer Datenbestände, ...
- **Benutzungsfreundlichkeit**: übersichtliche Oberfläche, verständliche Fehlermeldungen, Hilfetexte, ... - für die jeweilige Benutzergruppe
- **Benutzerdokumentation**: Vollständigkeit, Verständlichkeit, ...
- **Änderbarkeit, Wartbarkeit**: modulare Systemstruktur, verständliche Entwickler-Dokumentation, ...
- ...

Akzeptanztest (Abnahmetest):

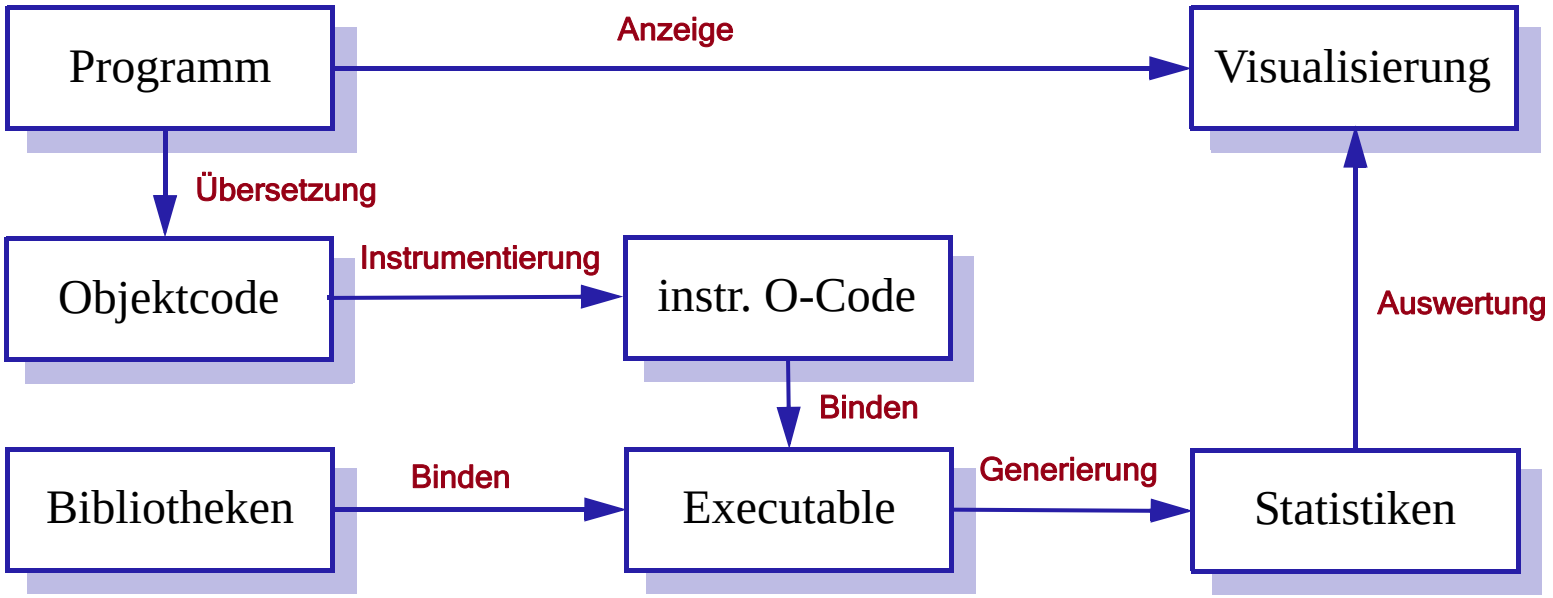
- Es handelt sich um eine **spezielle Form des Systemtests**
  - ⇨ der Kunde ist mit einbezogen bzw. führt den Test durch
  - ⇨ der Test findet beim Kunden, aber in Testumgebung statt (Test in Produktionsumgebung zu gefährlich)
- auf Basis des Abnahmetests entscheidet Kunde, ob das bestellte Softwaresystem **mangelfrei** ist und die im Lastenheft festgelegten Anforderungen erfüllt
- die durchgeführten Testfälle sollten bereits im **Vertrag** mit dem Kunden spezifiziert sein
- im Rahmen des Abnahmetests wird geprüft, ob System von allen relevanten Anwendergruppen **akzeptiert** wird
- im Rahmen von sogenannten **Feldtests** wird darüber hinaus ggf. das System in verschiedenen Produktionsumgebungen getestet

Testen, testen, … - wann ist Schluss damit?

- **nie** - nach jeder Programmänderung wird eine große Anzahl von Testfällen automatisch ausgeführt (siehe Regressionstest)
- Testbudget verbraucht bzw. Auslieferungszeitpunkt für Software erreicht (der Kunde testet unfreiwillig weiter ...)
- je Testfall (Zeiteinheit) gefundene Fehlerzahl sinkt unter gegebene Grenze (in der Hoffnung, dass die Anzahl der im Programm verbliebenen Fehler mit der Anzahl der pro Zeiteinheit gefundenen Fehler korreliert)
- n % absichtlich von einer Gruppe implantierter Fehler (seeded bugs) wurden von Testgruppe gefunden (siehe auch Mutationstestverfahren)
- gemäß systematischen Verfahren werden aus allen möglichen Eingabedatenkombinationen typische Repräsentanten ausgewählt und genau diese getestet (siehe Abschnitt 4.3)
- Testfälle decken hinreichend viele (relevante) Programmdurchläufe ab (siehe Testüberdeckungsmetriken in Abschnitt 4.4 und Abschnitt 4.5)

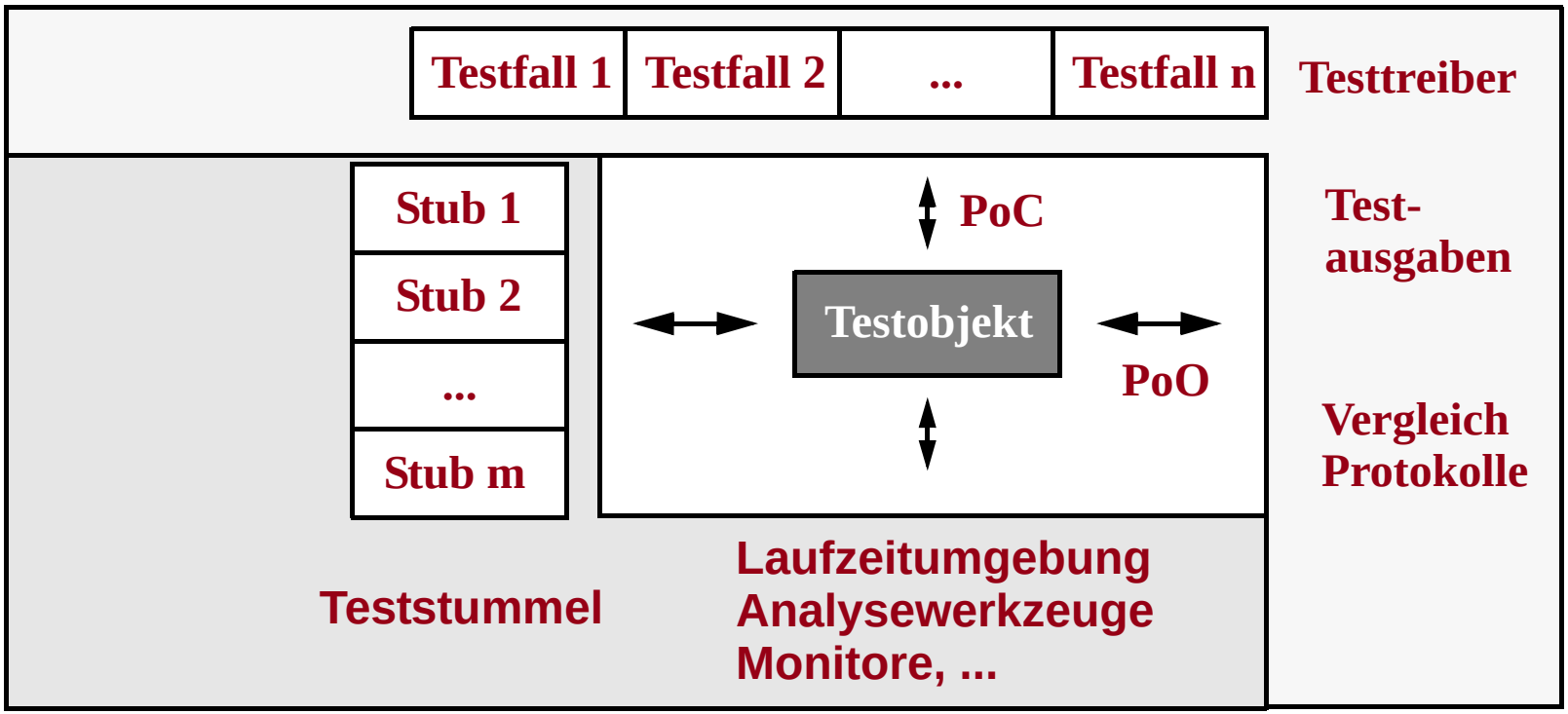
4.2 Laufzeit- und Speicherplatzverbrauchsmessungen

Gemeinsame Eigenschaft aller hier vorgestellten Werkzeuge/Verfahren ist:



- ⇨ Objektcode für untersuchte Software wird vor Ausführung „instrumentiert“ (um zusätzliche Anweisungen ergänzt)
- ⇨ zusätzliche Anweisungen erzeugen während der Ausführung statistische Daten über Laufzeitverhalten, Speicherplatzverbrauch, ...

Aufbau eines Testrahmens gemäß [SL12]:



- **Point of Control (PoC)**:
  Schnittstelle, über die Testobjekt mit Testdaten versorgt wird‘
- **Point of Observation (PoO)**:
  Schnittstelle, über die Reaktionen/Ausgaben des Testobjekts beobachtet werden

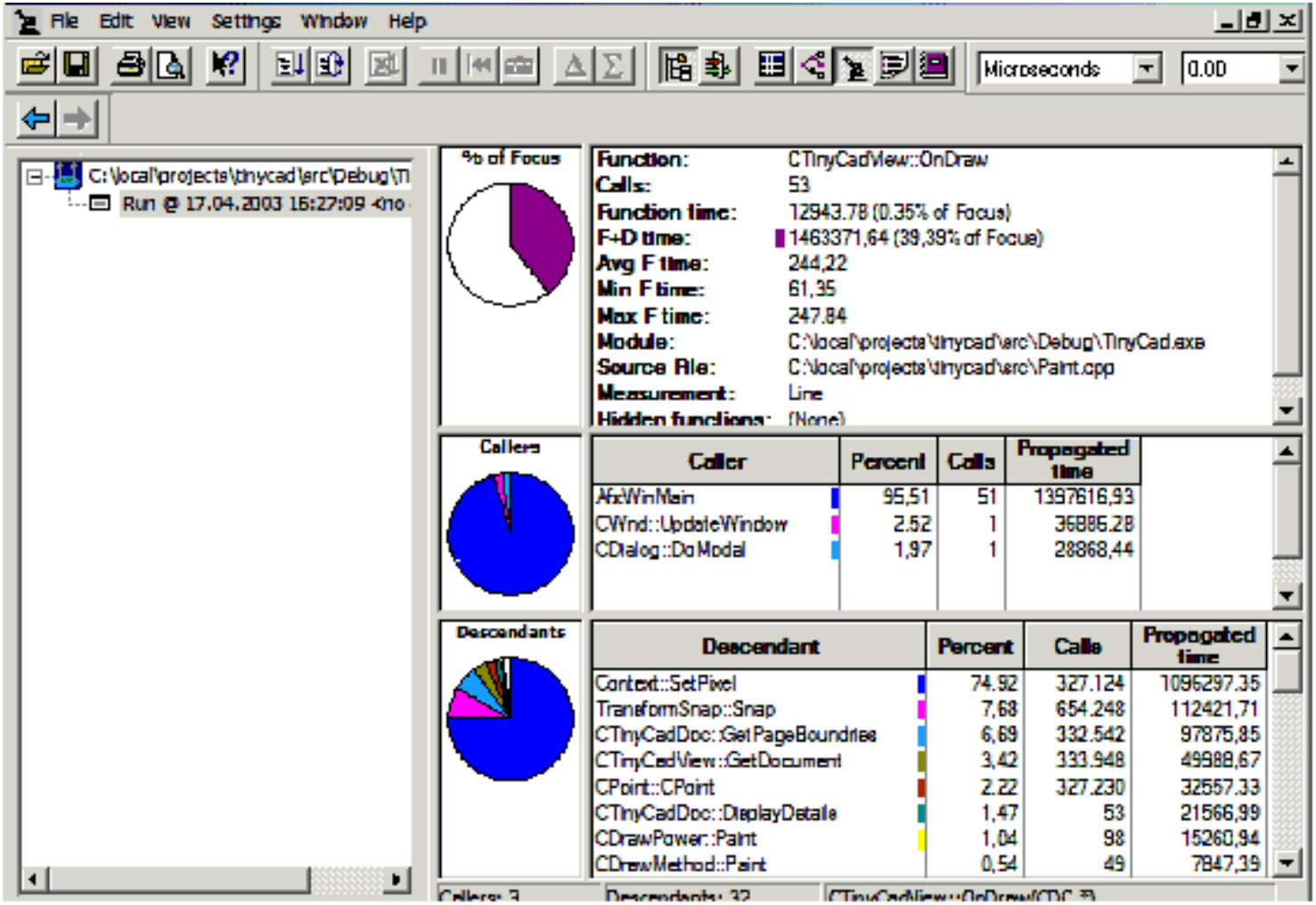
Untersuchung des Laufzeitverhaltens eines Programms:

- wie oft wird jede Operation aufgerufen (oder Quellcodezeile durchlaufen)
- welche Operation ruft wie oft welche andere Operation (**descendants**) auf oder von welchen Operationen (**callers**) wird ein Programm wie oft gerufen
- wieviel Prozent der Gesamtlaufzeit wird mit Ausführung einer bestimmten Operation verbracht (ggf. aufgeteilt nach callers und descendants)
- ...

Nutzen der ermittelten Daten:

- Operationen, die am meisten Laufzeit in Anspruch nehmen, können leicht identifiziert (und optimiert) werden
- tatsächliche Aufrufabhängigkeiten werden sofort sichtbar
- ...

Laufzeitanalyse mit Quantify (Rational/IBM):



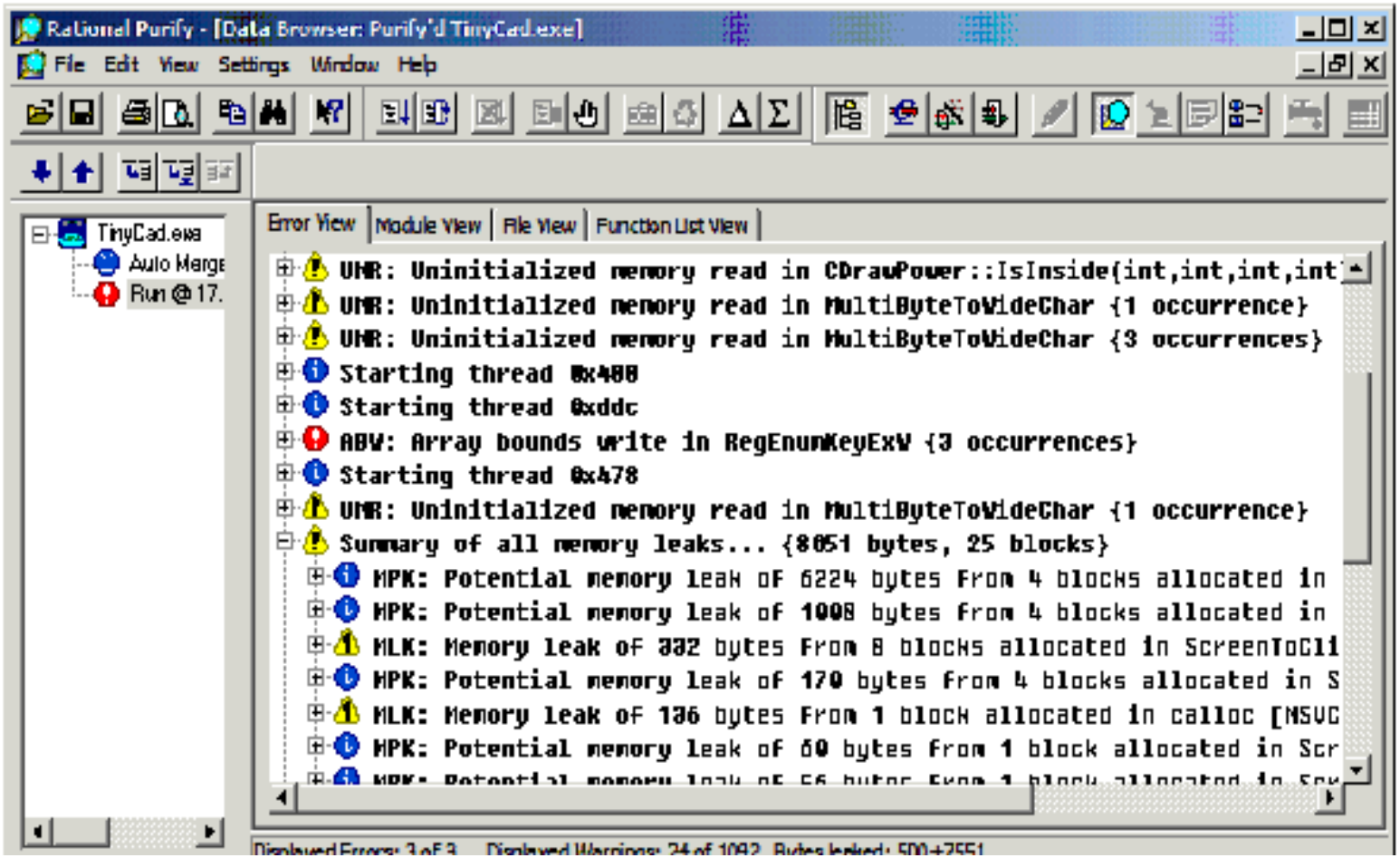
Erläuterungen zu Quantify-Beispiel:

- untersucht wird die Methode (Operation, Funktion) OnDraw
- im untersuchten Testlauf wurde OnDraw 53 mal aufgerufen
- die Ausführung der Methode OnDraw (inklusive gerufener Methoden) braucht 39% der Gesamtlaufzeit
- der Code der Methode OnDraw selbst benötigt aber nur 0,39% der Laufzeit
- die Ausführungszeit variiert sehr stark (61 bis 247 ms), Durchschnitt liegt aber nahe beim Maximum mit 244 ms
- die Methode OnDraw wird von Main 51 mal sowie von UpdateWindow und DoModal jeweils einmal im Beispiellauf gerufen
- die Methode OnDraw ruft ihrerseits die Methoden SetPixel, Snap, ...
- am meisten Zeit benötigen die 327.124 Aufrufe von SetPixel, nämlich fast 75% der Zeit

Untersuchung des Speicherplatzverhaltens eines Programms:

- welche Operationen fordern wieviel Speicherplatz an (geben ihn frei)
- wo wird Freigabe von Speicherplatz vermutlich bzw. bestimmt vergessen (memory leak = Speicherloch):
  - ⇨ bestimmt vergessen: Objekt lebt noch, kann aber nicht mehr erreicht werden (Garbage Collector von Java würde es entsorgen)
  - ⇨ vermutlich vergessen: Objekt lebt noch und ist erreichbar, wird aber nicht mehr benutzt (Garbage Collector von Java kann es nicht freigeben)
- wo wird auf bereits freigegebenen Speicherplatz zugegriffen (nur für C++) bzw. wo wird Speicherplatz mehrfach freigegeben (nur für C++)
- wo wird auf nicht initialisierten Speicherplatz zugegriffen; anders als bei der statischen Programmanalyse wird für jede Feldkomponente (Speicherzelle) getrennt Buch darüber geführt
- wo finden Zugriffe jenseits der Grenzen von Arrays statt (Laufzeitfehler in guter Programmiersprache)

Analyse des Speicherplatzverhalten mit Purify (Rational/IBM):

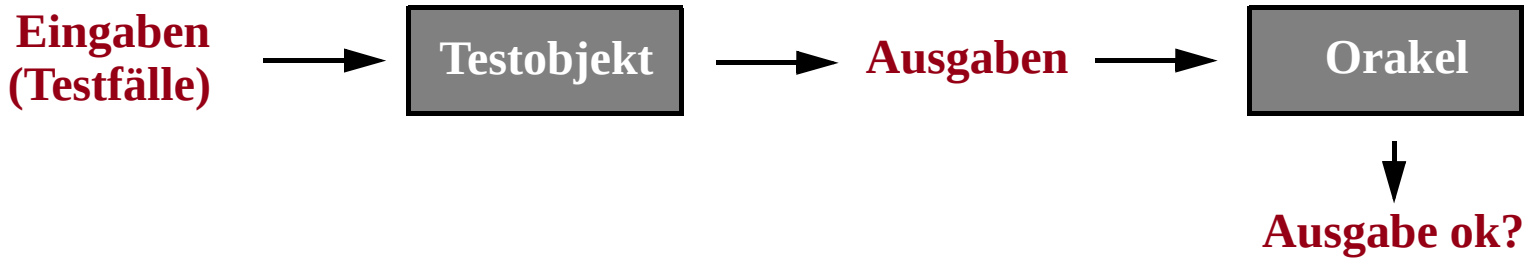

Rational Purify - [Data Browser: Purify'd TinyCad.exe]
File Edit View Settings Window Help
TinyCad.exe
Auto Merge
Run @ 17.
Error View
Module View
File View
Function List View
UMR: Uninitialized memory read in CDrawPower::IsInside(int,int,int,int
UMR: Uninitialized memory read in MultiByteToWideChar {1 occurrence}
UMR: Uninitialized memory read in MultiByteToWideChar {3 occurrences}
Starting thread 0x488
Starting thread 0xddc
ABW: Array bounds write in RegEnumKeyExW {3 occurrences}
Starting thread 0x478
UMR: Uninitialized memory read in MultiByteToWideChar {1 occurrence}
Summary of all memory leaks... {8051 bytes, 25 blocks}
MPK: Potential memory leak of 6224 bytes from 4 blocks allocated in
MPK: Potential memory leak of 1008 bytes from 4 blocks allocated in
MLK: Memory leak of 332 bytes from 8 blocks allocated in ScreenToCli
MPK: Potential memory leak of 170 bytes from 4 blocks allocated in S
MLK: Memory leak of 136 bytes from 1 block allocated in calloc [MSVC
MPK: Potential memory leak of 80 bytes from 1 block allocated in Scr

Achtung: so viele Speicherlöcher, ... ist typisch für Programmierung in C++

4.3 Funktionsorientierte Testverfahren

Sie testen Implementierung gegen ihre Spezifikation und lassen die interne Programmstruktur unberücksichtigt (Programm wird als „Black-Box“ behandelt):

- für **Abnahmetest** ohne Kenntnis des Quellcodes geeignet
- setzt (eigentlich) vollständige und widerspruchsfreie **Spezifikation** voraus (zur Auswahl von Testdaten und Interpretation von Testergebnissen)
- **repräsentative Eingabewerte** müssen ausgewählt werden (man kann im allgemeinen nicht alle Eingabekombinationen testen)
- man braucht „**Orakel**“ für Überprüfung der Korrektheit der Ausgaben (braucht man allerdings bei allen Testverfahren)



Kriterien für die Auswahl von Testdaten:

An der Spezifikation orientierte **Äquivalenzklassenbildung**, so dass für alle Werte einer Äquivalenzklasse (Eingabewertklasse) sich das Softwareprodukt „gleich“ verhält:

- Unterteilung in Klassen von Eingabewerten, für die das Programm sich laut Spezifikation gleich verhalten muss
- Alle Klassen von Eingabewerten zusammen müssen den ganzen möglichen Eingabewertebereich des betrachteten Programms abdecken
- Aus jeder Äquivalenzklasse wird mindestens ein repräsentativer Wert getestet
- Unterteilung in gültige und ungültige Eingabewerte (fehlerhafte Eingaben, ...) wird durchgeführt und später bei der Auswahl von Testwerten berücksichtigt
- Oft gibt es auch eine gesonderte Betrachtung von Äquivalenzklassen für besonders „große“ oder besonders „kleine“ gültige Eingaben (Stresstest)
- Hier nicht mit betrachtet wird die Problematik der Suche nach Eingabewerte-klassen, die zu bestimmten (Klassen von) Ausgabewerten führen

Regeln für die Festlegung von Eingabewerteklassen:

- für geordnete Wertebereiche:
  - ⇨]uv .. ov[ist ein offenes Intervall aller Werte zwischen uv und ov (uv und ov selbst gehören nicht dazu)
  - ⇨ [uv .. ov] ist ein geschlossenes Intervall aller Werte zwischen uv und ov (uv und ov selbst gehören dazu)
  - ⇨ Mischformen]uv .. ov] und [uv .. ov[sind natürlich erlaubt
- für ganze Zahlen (Integer):
  - ⇨ [MinInt .. ov] für Intervalle mit kleinster darstellbarer Integer-Zahl
  - ⇨ [uv .. MaxInt] für Intervalle mit größter darstellbarer Integer-Zahl
  - ⇨ offene Intervallgrenzen sind natürlich erlaubt
- für reelle Zahlen (Float) mit Voraussetzung kleinste/größte Zahl nicht darstellbar:
  - ⇨]- ∞ .. ov] oder]- ∞ .. ov[für nach unten offene Intervalle
  - ⇨ [uv .. ∞ [oder]uv .. ∞ [für nach oben offene Intervalle
  - ⇨ alle Mischformen von Intervallen mit festen unteren und oberen Grenzen

Regeln für die Festlegung von Eingabewerteklassen - Fortsetzung:

- für Zeichenketten (String):
  - ⇨ Definition über reguläre Ausdrücke oder Grammatiken
  - ⇨ Aufzählung konkreter Werte (siehe nächster Punkt)
- für beliebige Wertebereiche:
  - ⇨ {v1 v2 v3 ... vn} für Auswahl von genau n verschiedenen Werten
- für zusammengesetzte Wertebereiche:
  - ⇨ Anwendung der obigen Prinzipien auf die einzelnen Wertebereiche
  - ⇨ ggf. braucht man noch zusätzliche Einschränkungen (Constraints), die nur bestimmte Wertekombinationen für die Teilkomponenten zulassen
- Eingabewerteklassen, die aus genau einem Wert bestehen:
  - ⇨ [v] es ist aber auch die Repräsentation {v} oder v üblich

Auswahl von Testdaten aus Eingabewerteklassen:

- aus jeder Eingabewerteklasse wird mindestens ein Wert ausgewählt (Fehler- und Stressklassen werden später gesondert behandelt)
- gewählt werden meist nicht nur die Grenzen selbst, sondern auch die um eins größeren und kleineren Werte (siehe ISTQB-Prüfung; im Folgenden werden aus Platzgründen bei den Beispielen aber nur Grenzwerte selbst ausgewählt)
- als Intervalle dargestellte Eingabewerteklassen werden oft durch die so genannte **Grenzwertanalyse** nochmal in Unterklassen/Teilintervalle zerlegt:
  - ⇨]uv .. ov[wird zerlegt in [inc(uv)]]inc(uv) .. dec(ov)[[dec(ov)]
  - ⇨ [uv .. ov] wird zerlegt in [uv]]uv .. ov[[ov]
  - ⇨ ...
  - ⇨ inc(uv) liefert den nächstgrößeren Wert zu uv
  - ⇨ dec(ov) liefert den nächstkleineren Wert zu ov
  - ⇨ Achtung: bei Float muss für inc und dec die gewünschte Genauigkeit festgelegt werden, mit der aufeinanderfolgende Werte gewählt werden

Zusätzliche Wahl von Testdaten durch Grenzwertanalyse:

Bilden Eingabewerteklassen einen Bereich (Intervall), so selektiert die Grenzwert-analyse also immer Werte um die Bereichsgrenzen herum:

- gewählt werden meist nicht nur die Grenzen selbst, sondern auch die um eins größeren und kleineren Werte (im Folgenden aus Platzgründen weggelassen)
- Idee dabei: oft werden Schleifen über Intervallen gebildet, die genau für die Grenzfälle falsch programmiert sind
- Beispiel für die Grenzwertanalyse:
  zulässiger Eingabebereich besteht aus reellen Zahlen (Float) mit zwei zu betrachtenden Nachkommastellen zwischen -1 und +1 (Fehlerklassen in **Rot**):
  - ⇨ vor Grenzwertanalyse:
    ]- ∞ .. -1.0[[-1.0 .. +1.0]]+1.0 .. + ∞ [
  - ⇨ nach Grenzwertanalyse:
    ]- ∞ .. -1.01[[-1.01] [-1.0]]-1.0 .. +1.0[[+1.0] [+1.01]]+1.01 .. + ∞ [
  - ⇨ Beispiele für gewählte Werte: -2, -1.01, -1.0, 0, +1.0, +1.01, +2

Auswahl von Testeingaben für countVowels (mit Zeichenketten!):

PROCEDURE countVowels(s: Sentence; VAR count: INTEGER);
(Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)*

- Klasse 1: s endet nicht mit einem Punkt (oh je, das geht schief)
- Klasse 2: s endet mit einem Punkt und enthält keine Vokale
  - ⇨ s besteht nur aus einem Punkt
  - ⇨ s besteht aus einem Konsonanten gefolgt von einem Punkt
  - ⇨ s enthält sonstige Sonderzeichen (wie etwa !"§$%&/()=?)
- Klasse 3: s endet mit einem Punkt und enthält einen Vokal:
  - ⇨ 3a: s enthält ein a, e, i, o, u
  - ⇨ 3b: s enthält ein A, E, I, O, U (oh je, dieser Fall wurde auch vergessen)
- Klasse 4: s enthält mehrere Vokale:
  - ⇨ 4a: mehrere gleiche Vokale
  - ⇨ 4b: mehrere verschiedene Vokale
- Klasse 5: Eingabe ist sehr lang und enthält ganz viele Vokale

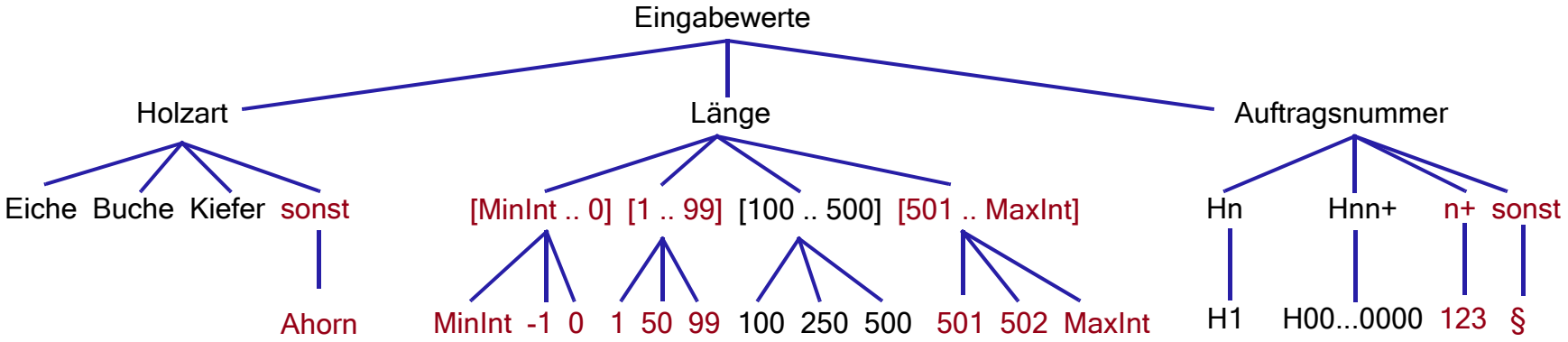
Beispiel Lagerverwaltung einer Baustoffhandlung - Holzbretter:

- bei Holzbrettern wird die Holzart eingegeben (als String)
- Eiche, Kiefer und Buche sind bekannt
- es wird die Brettlänge eingegeben (zwischen 100 und 500 cm)
- jede Lieferung erhält eine Auftragsnummer, die mit „H“ beginnt und eine Länge von 1 bis 20 Zeichen hat
- aus Brettlänge und Holzart errechnet sich der Preis des Auftrags

Weitere Regeln für die Bildung von Äquivalenzklassen bei Intervallen:

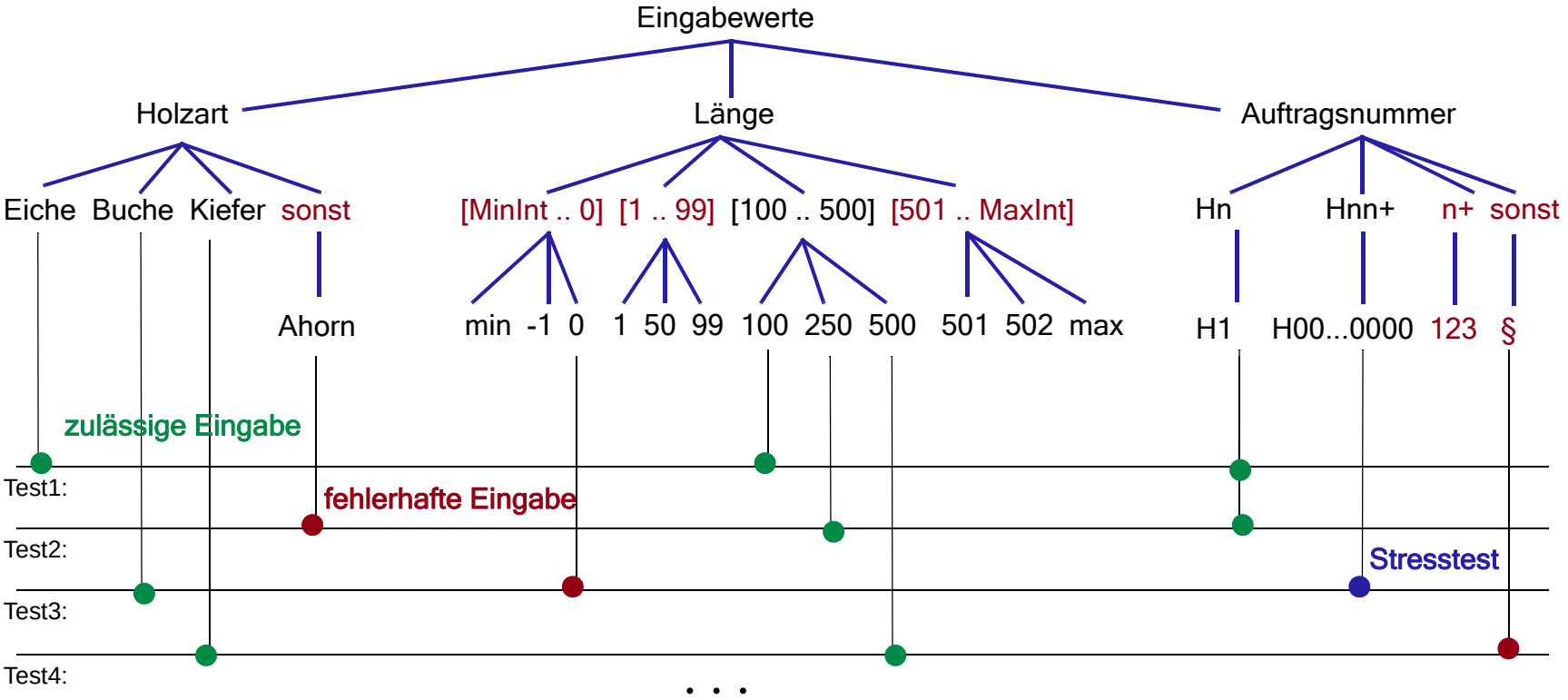
- aus Äquivalenzklassen, die (geschlossene) Intervalle sind, werden jeweils die beiden Grenzwerte und ein weiterer Wert (z.B. aus der Mitte des Intervalls) ausgewählt (ohne platzraubenden Zwischenschritt der Zerlegung in drei Teilintervalle)
- einelementige Äquivalenzklasse und Wert aus dieser Äquivalenzklasse werden nicht unterschieden, also statt [v] schreiben wir gleich v
- reguläre Ausdrücke werden zur Definition v. String-Äquivalenzklassen eingesetzt

Grafische Darstellung von Äquivalenzklassen als Klassifikationsbaum:



- zunächst wird „Eingabewerte“ in alle Eingabeparameter des Programms zerlegt
- zusammengesetzte Eingabeparameter werden weiter zerlegt (nicht im Beispiel)
- schließlich wird der Wertebereich eines atomaren Eingabeparameters betrachtet
- der Wertebereich wird in Äquivalenzklassen „ähnlicher“ Werte zerlegt
- dabei werden auch nicht erlaubte Eingabewerte (Fehlerklassen) betrachtet
- ebenso werden „extreme“ erlaubte Werte (Stresstestklassen) betrachtet
- aus jedem Wertebereich werden Repräsentanten für den Test ausgewählt

Unvollständige/fehlerhafte Auswahl von Eingabewertekombinationen:



Problem: trotz Bildung von Äquivalenzklassen bleiben (zu) viele mögliche Eingabewertekombinationen (im Beispiel sind 4 * 12 * 4 verschiedene Testläufe möglich).

Heuristiken für die Reduktion möglicher Eingabewertekombinationen:

- aus jeder **Äquivalenzklasse** wird *mindestens* einmal ein Wert ausgewählt; es gibt also jeweils mindestens einen Testfall, in dem ein Eingabewert der Äquivalenz-klasse verwendet wird (bei Grenzwertanalyse werden drei Werte ausgewählt)
- bei **abhängigen Eingabeparametern** müssen Testfälle für **alle Kombinationen** ihrer jeweiligen „normalen“ Äquivalenzklassen aufgestellt werden; Parameter sind abhängig, wenn sie gemeinsam das Verhalten des Programms steuern (und deshalb nicht unabhängig voneinander betrachtet werden können)
- der ausgewählte **Wert** einer **Fehleräquivalenzklasse** wird in genau einem Testfall verwendet (Fehleräquivalenzklassen sind solche Äquivalenzklassen, die unzuläs-sige Eingabewerte zusammenfassen)
- der ausgewählte **Wert** einer **Stresstestklasse** wird ebenfalls genau einmal in einem Testfall verwendet (Stresstestklassen sind solche Äquivalenzklassen, die beson-ders „große/lange/...“ zulässige/gültige Eingabewerte zusammenfassen)
- hat man mehrere Eingabeparameter, wird **höchstens einer** mit einem Wert aus einer Fehler- oder Stresstestklasse belegt (verhindert Verdeckung von Fehlern)

Sinnvolle Auswahl von Testfällen am Beispiel demonstriert:

- die fehlerhaften Auftragsnummern H und 123 treten nur in jeweils einem Testfall auf, die fehlerhafte Eingabe Ahorn ebenfalls
- gleiches gilt für eine zu kleine und eine zu große Längenangabe
- die extrem lange Auftragsnummer H00...0000 tritt ebenfalls nur einmal in einem Testfall auf
- zu kleine und zu große Längenangaben, fehlerhafte Auftragsnummern, falsche Holzarten und die extrem lange Auftragsnummer treten nicht im selben Testfall auf
- alle zulässigen Kombinationen von Äquivalenzklassen für Holzarten und Längen-angaben müssen durchgetestet werden, da laut Spezifikation Holzart und Länge gemeinsam den Preis des Auftrags bestimmen
- da Auftragsnummern laut Spezifikation der Software unabhängig von Holzarten und Längen verarbeitet werden, wird jede Kombination von Holzart und Länge nur mit einer normalen Auftragsnummer getestet

Geändertes Beispiel für Testparameterauswahl:

Wieder soll eine Vorschrift zur Berechnung des Preises von Brettern getestet werden. Die Eingabeparameter sind dieses Mal:

- Holzart (Aufzählungstyp): Buche, Eiche, Kiefer
- Brettlänge (Zahl größer gleich Null) mit folgenden Intervallen, in denen unterschiedliche Berechnungsschemata verwendet werden:
  kurze Bretter mit [0 .. 99] cm
  normale Bretter mit [100 .. 300] cm
  (zu) lange Bretter mit]300 .. + ∞ [cm
- Auftragsart (Aufzählungstyp): DoItYourself, Normal, Express

Alle drei Parameter interagieren bei der Berechnung des Preises eines Bretts; also müssten eigentlich zumindest alle

3 x 3 x 3 = 27 Kombinationen v. Äquivalenzklassen

getestet werden.

Paarweiser Testansatz (Pairwise Testing):

Die Praxis zeigt, dass ca. 80% aller von bestimmten Parameterwertkombinationen ausgelösten Softwarefehler bereits durch Wahl bestimmter Paarkombinationen beobachtet werden können. Also werden beim „paarweisen“ Testen einer Funktion mit n Parametern nicht alle möglichen Kombinationen überprüft, sondern nur alle paarweisen Kombinationen.

Beispiel:

Holzart	Buche	Buche	Buche	Eiche	Eiche	Eiche	Kiefer	Kiefer	Kiefer
Länge	kurz	norm	lang.	kurz	norm	lang.	kurz	norm	lang
Auftragsart	DoIt	Norm	Expr.	Expr.	DoIt	Norm	Norm	Expr.	DoIt

Im Beispiel werden für die Erzeugung aller möglichen paarweisen Kombinationen von Äquivalenzklassen von Testwerten nur 9 Testfälle (Testvektoren) benötigt anstelle der 27 möglichen 3er-Kombinationen!

Bewertung der funktionalen Äquivalenzklassenbildung:

- Güte des Verfahrens hängt stark von **Güte der Spezifikation** ab (Aussagen über erwartete/zulässige Eingabewerte und Ergebnisse)
- Verwaltung von Äquivalenzklassen und Auswahl von Testfällen kann durch **Werkzeugunterstützung** vereinfacht werden
- **ausführbare Modelle** (z.B. in UML erstellt) können bei der Auswahl von Äquivalenzklasssen, Testfällen helfen (und als Orakel verwendet werden)
- Test von Benutzeroberflächen, zeitlichen Beschränkungen, Speicherplatz-beschränkungen etc. (**nichtfunktionale Anforderungen**) wird kaum unterstützt
- mindestens **einmalige Ausführung** jeder Programmzeile wird nicht garantiert
- Test **zustandsbasierter Software** (Ausgabeverhalten hängt nicht nur von Eingabe, sondern auch von internem Zustand ab) geht so nicht
  - ⇨ später Spezifikation und Testplanung mit Hilfe von Automaten/Statecharts
  - ⇨ Grenzfall zwischen funktionalem und strukturorientiertem Softwaretest

Weitere Black-Box-Testverfahren:

- [**intuitives Testen**: aus dem Bauch heraus zusätzliche Testfälle festlegen]
- **Zufallstest**: wählt aus Menge der möglichen Werte eines Eingabedatums zufällig Repräsentanten aus (ggf. gemäß bekannter statistischer Verteilung)
  - ⇨ zur Ergänzung gut geeignet; es werden ggf. ganz „unerwartete“ Testdaten ausgewählt
- **Smoke-Test**: es wird nur Robustheit des Testobjekts getestet, berechnete Ausgabe-werte spielen keine Rolle (auf Tastatur hämmern, ...)
- **Syntax-Test**: ist für Eingabewerte der erlaubte syntaktische Aufbau bekannt (als Grammatik angegeben) kann man daraus systematisch Testfälle generieren
  - ⇨ Beispiele: zulässige Email-Adresse, ganze Zahl, Float,
  - ⇨ Pfadangaben von Unterverzeichnissen, ...
- **Zustandsbezogener Test**: siehe Abschnitt 4.6
- **Ursache-Wirkungs-Graph-Analyse**: siehe folgende Folien
- **Anwendungsfallbasiertes Testen**: siehe folgende Folien

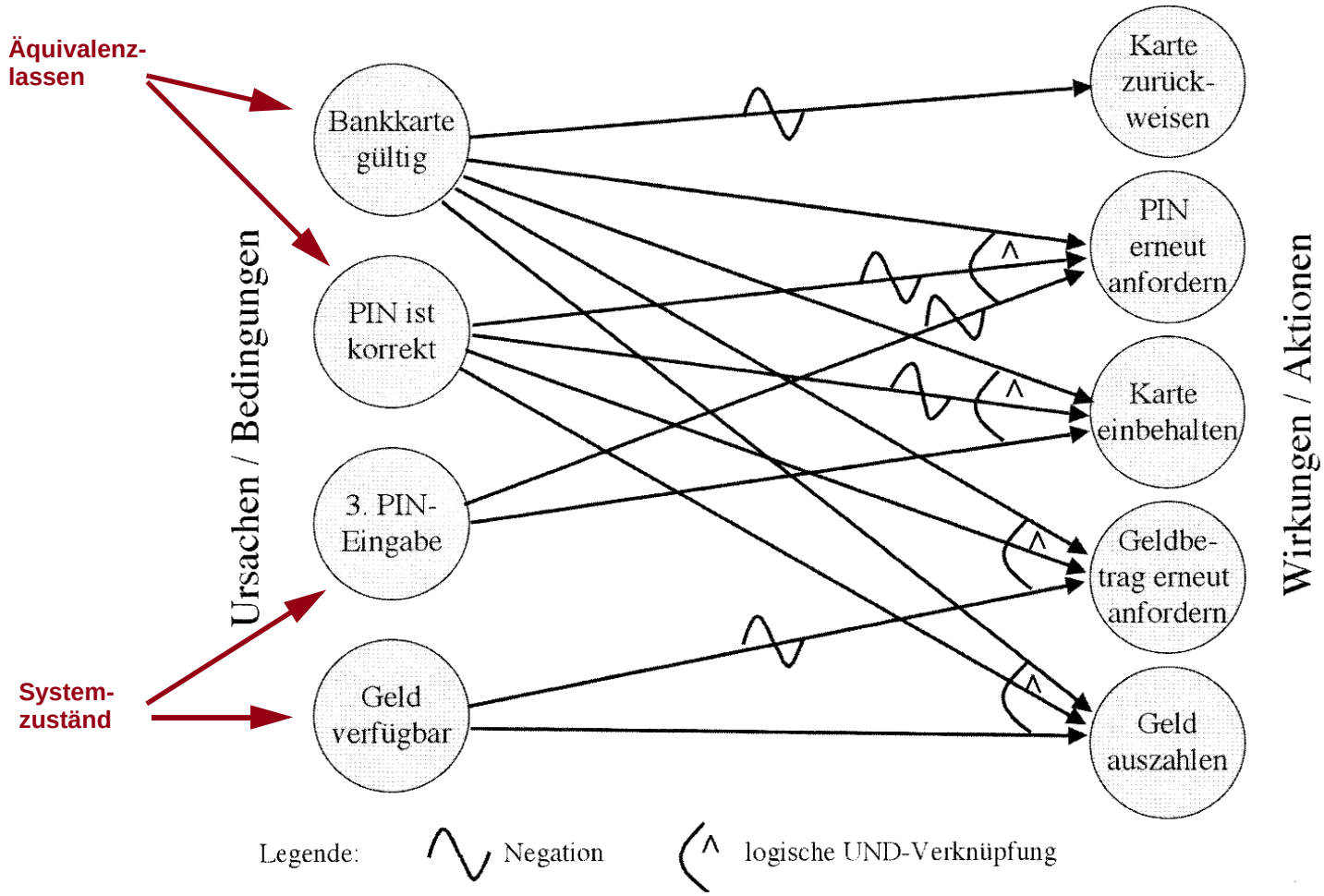
Ursache-Wirkungs-Graph-Analyse-Verfahren aus [SL12]:

Es handelt sich eigentlich um eine Kombination von zwei Verfahren. Die Basis bilden sogenannte „**Entscheidungstabellen**“, die Bedingungen an Eingaben und ausgelöste Aktionen eines Systems miteinander verknüpfen. Für die systematische Erstellung einer Entscheidungstabelle wird zunächst ein „**Ursache-Wirkungs-Graph**“ erstellt.

Ein Ursache-Wirkungs-Graph verknüpft Eingaben = **Ursachen** / Bedingungen mit daraus resultierenden Ausgaben = **Wirkungen** / Aktionen (durch grafische Darstellung aussagenlogischer Ausdrücke). Die weitere Vorgehensweise ist wie folgt:

1. Eine Wirkung wird ausgewählt.
2. Zu der Wirkung werden alle Kombinationen von Ursachen gesucht, die diese Wirkung hervorrufen.
3. Für jede gefundene Ursachenkombination wird eine Spalte der Entscheidungstabelle erzeugt.
4. Die Spalten der Entscheidungstabelle entsprechen Testfällen.

Ursache-Wirkungs-Graph-Beispiel aus [SL12]:



Entscheidungstabellen-Beispiel aus [SL12]:

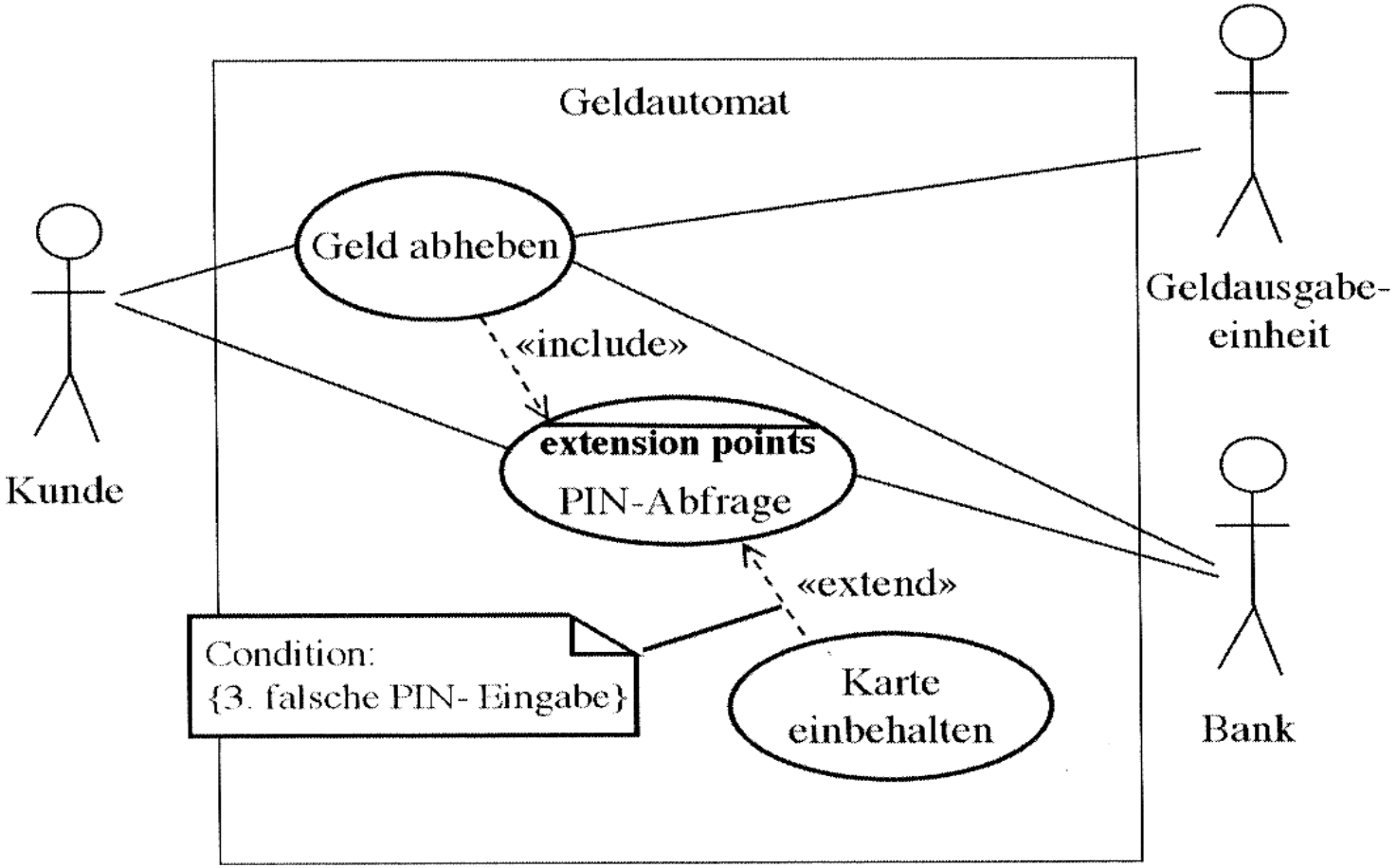
Entscheidungstabelle		TF1	TF2	TF3	TF4	TF5
Bedingungen	Bankkarte gültig?	Nein	Ja	Ja	Ja	Ja
	PIN ist korrekt?	–	Nein	Nein	Ja	Ja
	Dritte PIN-Eingabe?	–	Nein	Ja	–	–
	Geld verfügbar?	–	–	–	Nein	Ja
Aktionen	Karte zurückweisen	Ja	Nein	Nein	Nein	Nein
	PIN erneut anfordern	Nein	Ja	Nein	Nein	Nein
	Karte einbehalten	Nein	Nein	Ja	Nein	Nein
	Geldbetrag erneut anfordern	Nein	Nein	Nein	Ja	Nein
	Geld auszahlen	Nein	Nein	Nein	Nein	Ja

Anwendungsfallbasiertes Testen aus [SL12]:

Ausgangspunkt für diese Testmethodik ist die Beschreibung von sogenannten **Anwendungsfällen** (Nutzungsszenarien, Geschäftsvorfällen) eines Systems im Zuge der Anforderungsanalyse. Dabei kommt in der Regel die „Unified Modeling Language“ (UML) mit ihren Anwendungsfalldiagrammen zum Einsatz (siehe etwa Vorlesung „Software-Engineering - Einführung“):

- Anwendungsfalldiagramme erlauben die Beschreibung von Nutzungsszenarien (und damit von Akzeptanztestfällen) auf sehr hohem Abstraktionsniveau sowie die Unterscheidung zwischen normalen Abläufen und Ausnahmen.
- Werden einzelne Anwendungsfälle informell / in natürlicher Sprache beschrieben, so werden die dazugehörigen Testfälle „**manuell**“ erstellt.
- Werden für die Beschreibung einzelner Anwendungsfälle formalere Notationen wie Sequenz- oder Aktivitätsdiagramme der UML eingesetzt, so können Testwerkzeuge daraus **automatisch** Code für die Testfallausführung und -bewertung generieren.

Anwendungsfalldiagramm-Beispiel aus [SL12]:



Bewertung der Ursache-Wirkungs-Graph- und Anwendungsfall-Testens:

- Beide Methoden lassen sich sehr früh im Software-Lebenszyklus einsetzen und eignen sich insbesondere für die Erstellung von Akzeptanztestfällen.
- Die Ursache-Wirkungsgraph-Methode erlaubt die bei der Äquivalenzklassen-Bildung fehlende Verknüpfung von Eingaben und Ausgaben (Ursachen und Wirkungen).
- Das Anwendungsfallbasierte Testen erlaubt hingegen nicht nur die Beschreibung einzelner Testvektoren (Eingabewertkombinationen), sondern auch die Spezifikation ganzer Interaktionssequenzen zwischen Umgebung zu testendem System.
- Insbesondere das Anwendungsfallbasierte Testen unterstützt aber nicht die systematische Identifikation fehlender Testfälle (für bestimmte Eingabetestvektoren).

Fazit:

Alle vorgestellten „Black-Box“-Testmethoden (Funktionsorientierte Testverfahren) besitzen ihre Stärken und Schwächen und ergänzen einander!!!

4.4 Kontrollflussbasierte Testverfahren

Ausgangspunkt ist der Kontrollflussgraph einer Komponente; getestet werden sollen

```
PROCEDURE countVowels(s: Sentence; VAR count: INTEGER);
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN
   count := 0; i := 0;

   WHILE s[i] # '.' DO
      IF s[i]= 'b' OR s[i]= 'a' OR ...          (* Kontrollflussfehler - falsche Prüfung auf 'b'. *)
      THEN
         count := 1;                            (* Berechnungsfehler - count wird nicht erhöht. *)
         i := i+1;                              (* Kontrollflussfehler - nur bedingte Inkr. von i  *)
      END;
   END (* WHILE *)

END countVowels

   …

countVowels('to be . . . or not to be.', count);   (*Schnittstellenfehler - dot im Satz. *)
```

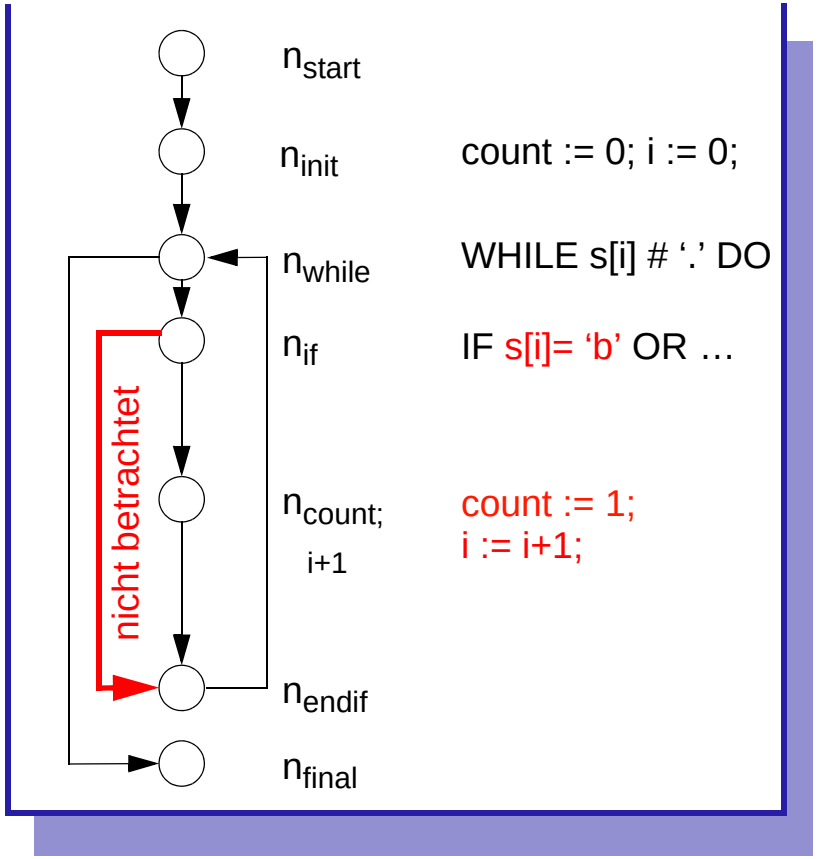
Grundideen des kontrollflussbasierten Testens:

- mit im Grunde zunächst beliebigen Verfahren werden **Testfälle festgelegt**
- diese Testfälle werden alle **ausgeführt** und dabei wird notiert, welche Teile des Programms durchlaufen wurden
- es gibt (meist) ein **Test-Orakel**, dass für jeden ausgeführten Testfall ermittelt, ob die berechnete Ausgabe (Verhalten des Programms) korrekt ist
- schließlich wird festgelegt, ob die vorhandenen Testfälle den Kontrollfluss des Programms hinreichend **überdecken**
- ggf. werden solange **neue Testfälle aufgestellt**, bis hinreichende Überdeckung des Quelltextes (Kontrollflusses) erreicht wurde
- ggf. werden **alte Testfälle gestrichen**, die die selben Teiles des Quelltextes (Kontrollflusses) überdecken

Kontrollflusstest - Anweisungsüberdeckung (C_0-Test):

Jeder Knoten des Kontrollflussgraphen muß mindestens einmal ausgeführt werden.

- Minimalkriterium, da nicht mal alle Kanten des Kontrollflussgraphen traversiert werden
- viele Fehler bleiben unentdeckt



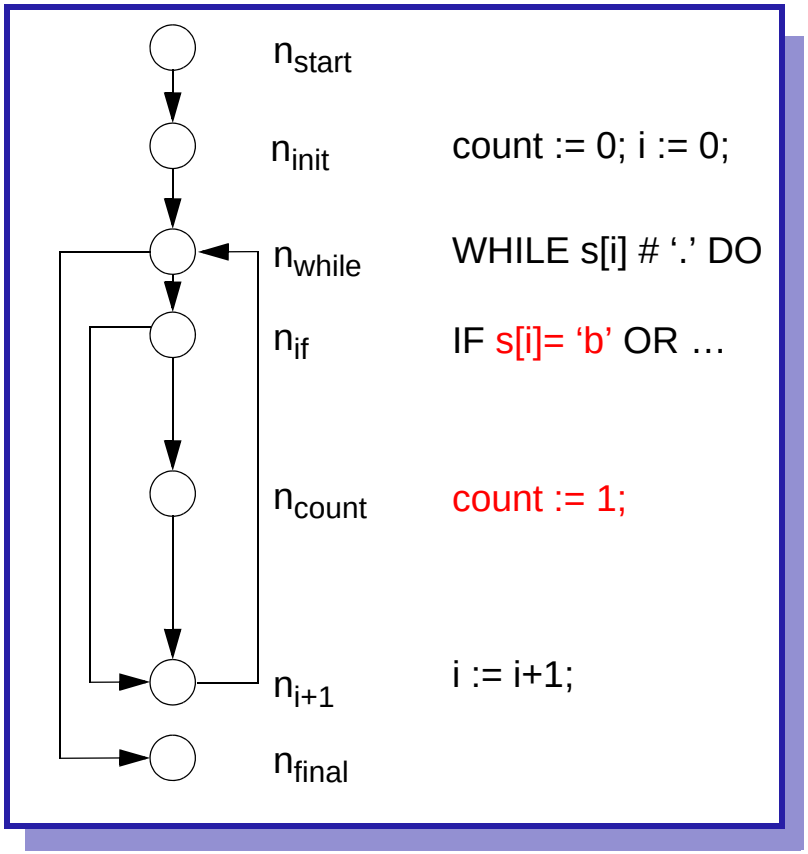
Beispiel:

- ☹ als Testfall genügt ein konsonantenfreier Satz wie etwa 'a.'
- ☹ Verschiebung von i := i+1; in if-Anweisung wird nicht entdeckt
- ☹ fehlerhafte Anweisung count := 1; wird nicht entdeckt
- ☹ fehlerhafte Bedingung wird nicht entdeckt

Kontrollflusstest - Zweigüberdeckung (C_1-Test):

Jede Kanten des Kontrollflussgraphen muß mindestens einmal ausgeführt werden.

- realistisches Minimalkriterium
- umfasst Anweisungsüberdeckung
- Fehler bei Wiederholung oder anderer Kombination von Zweigen bleiben unentdeckt



Beispiel:

- ☹ Testfall 'ax.' für countVowels würde Kriterium bereits erfüllen
- ☹ Zuweisung count := 1; wird nicht als fehlerhaft erkannt
- ☹ ebenso nicht die Bedingung s[i] = 'b'

Kontrollflusstest - Bedingungsüberdeckung:

Jede Teilbedingung einer Kontrollflussbedingung (z.B. von if- oder while-Anweisung) muß einmal den Wert true und einmal den Wert false annehmen.

atomare Bedingungsüberdeckung/-test:

- ⇨ keine Anforderung an Gesamtbedingung
- ⇨ bei falschem countVowels reicht Testfall ‘baeiou.’
- ⇨ umfasst nicht mal Anweisungsüberdeckung (Beispiel?)

minimale Mehrfachbedingungsüberdeckung/-test:

- ⇨ jede Teil- und Gesamtbedingung ist einmal true und einmal false
- ⇨ bei falschem countVowels reicht Testfall ‘xbaeiou.’
- ⇨ Orientierung an syntaktischer Struktur von Kontrollflussbedingungen
- ⇨ Bedingung umfasst Zweigüberdeckung
- ⇨ trotzdem werden viele Bedingungsfehler nicht entdeckt

Atomare Bedingungsüberdeckung/-test mit ‘baeiou.´:

```
PROCEDURE hasVowels(s: Sentence; VAR exists: BOOLEAN);
   (* Returns true iff sentence contains vowels. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN
   exists := FALSE; i := 0;

   WHILE s[i] # ‘.’ DO
      IF s[i]= ‘b’ OR s[i]= ‘a’ OR ...          (* Kontrollflussfehler - Prüfung auf ‘b’ falsch. *)
      THEN
         exists := TRUE;
         i := i+1;                              (* Kontrollflussfehler - nur bedingte Inkr. von i  *)
      END;
   END (* WHILE *)

END countVowels

   …

countVowels(‘baeiou.’, count);
```

In diesem Fall ist die if-Bedingung bei jedem Schleifendurchlauf „wahr“, also hat man keine Zweigüberdeckung. Fehler werden nicht entdeckt!

Minimale Mehrfachbedingungsüberdeckung/-test mit ‘xbaeiou.´:

```
PROCEDURE hasVowels(s: Sentence; VAR exists: BOOLEAN);
   (* Returns true iff sentence contains vowels. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN
   exists := FALSE; i := 0;

   WHILE s[i] # '.' OR NOT exists DO          (* Kontrollflussfehler - OR statt AND falsch. *)
      IF s[i]= 'b' OR s[i]= 'a' OR ...        (* Kontrollflussfehler - Prüfung auf 'b' falsch. *)
      THEN
         exists := TRUE;
      END;
      i := i+1;
   END (* WHILE *)

END countVowels

   …

countVowels('xbaeiou.', exists);
```

Man hat Zweigüberdeckung, Fehler werden trotzdem nicht gefunden! Besser wären hier viele kurze Eingaben mit **einem** Schleifendurchlauf anstelle einer langen Eingabe.

Modifizierter Bedingungsüberdeckungstest:

Der modifizierte Bedingungsüberdeckungstest (Definierter Bedingungstest, Minimaler Mehrfachbedingungstest) benötigt wie die bisherigen Überdeckungskriterien eine linear mit der Anzahl der atomaren Teilbedingungen steigende Anzahl von Testfällen. Es werden aber i.a. „bessere“ Testfälle als bei der minimalen Mehrfachbedingungs-überdeckung gewählt. Die Bedingungen sind:

- jeder atomaren Teilbedingung lassen sich zwei Testfälle zuordnen (verschiedene Teilbedingungen dürfen aber die selben Testfälle nutzen)
- die Werte aller Teilbedingungen, die wegen „short circuit“-Evaluation des Compilers von Booleschen Ausdrücken nicht ausgewertet werden, bleiben im Testfall undefiniert (und können deshalb beliebig gesetzt werden)
- die beiden Testfälle zu einer atomaren Teilbedingung setzen diese einmal auf true und einmal auf false und unterscheiden sich nur in der gerade betrachteten atomaren Teilbedingung (undefiniert kann mit true u. false gleichgesetzt werden)
- die beiden Testfälle zu einer atomaren Teilbedingung setzen die Gesamtbedingung einmal auf true und einmal auf false

Beispiele für „short circuit“-Evaluierung (von links nach rechts):

- IF <Ausdruck der true liefert> OR b THEN ... :
  - ⇨ die Belegung von b ist irrelevant für das Testen und wird in vielen Programmiersprachen deshalb nicht ausgewertet
- IF <Ausdruck der false liefert> AND b THEN ... :
  - ⇨ die Belegung von b ist irrelevant für das Testen und wird in vielen Programmiersprachen deshalb nicht ausgewertet

Einfaches Beispiel für modifzierten Bedingungsüberdeckungstest:

a	b	a OR b
0	0	0
1	-	1
0	1	1

Anmerkung: der erste Testfall wird für die Bedingung a und b genutzt.

Komplexeres Beispiel für Bedingungsüberdeckungsalternativen:

Art der Überdeckung	a	b	c	(a OR b) AND c
atomar	1	1	0	0
	0	0	1	0
Zweig	1	0	0	0
	1	0	1	1
min. mehrfach	0	0	0	0
	1	1	1	1
modifiziert mehrfach	0	0	-	0
	1	-	1	1
	0	1	1	1
	1	-	0	0

Kontrollflusstest - Pfadüberdeckung (C-„unendlich“-Test):

Jeder mögliche Pfad des Kontrollflussgraphen muss einmal durchlaufen werden.

- rein theoretisches Kriterium, sobald Programm Schleifen enthält (unendliche viele verschiedene Pfade = Programmdurchläufe möglich)
- dient als Vergleichsmaßstab für andere Testverfahren
- findet trotzdem nicht alle Fehler (z.B. Berechnungsfehler), da kein erschöpfender Test aller möglichen Eingabewerte (siehe Abschnitt 4.3)
- davon abgeleitetete in der Praxis durchführbare Verfahren:
  - ⇨ **boundary test**: *alle* Pfade auf denen Schleifen maximal einmal durchlaufen werden (ohne besondere praktische Bedeutung)
  - ⇨ **boundary interior test**: *alle* Pfade auf denen Schleifen maximal zweimal (in direkter Folge) durchlaufen werden (Achtung: Anzahl Pfade explodiert bei geschachtelten Schleifen und vielen bedingten Anweisungen)
  - ⇨ **modifizierter boundary interior test**: bei geschachtelten Schleifen wird beim Durchlauf einer äußeren Schleife die Anzahl der inneren Schleifen-durchläufe nicht unterschieden

Abstraktes Beispiel für Boundary-Interior-Test:

```
IF b1 THEN n1 ELSE n2 END;

WHILE b2 DO
   n3;
   WHILE b3 DO
      IF b4 THEN n4 ELSE n5 END;
   END;
END;
```

Durchläufe für Boundary-Interior-Test:

n1	n2	(* kein Durchlauf *)
n1, n3	n2, n3	(* 1x äussere, 0x innere Schleife *)
n1, n3, n4	n2, n3, n4	(* 1x äussere, 1x innere Schleife *)
n1, n3, n5	n2, n3, n5	
n1, n3, n4, n4	n2, n3, n4, n4	(* 1x äussere, 2x innere Schleife *)
n1, n3, n4, n5	n2, n3, n4, n5	
n1, n3, n5, n4	n2, n3, n5, n4	
n1, n3, n5, n5	n2, n3, n5, n5	
n1, n3, n3	n2, n3, n3	(* 2x äussere, 0x innere Schleife *)
n1, n3, n4, n3	n2, n3, n4, n3	(* 2x äussere, 1x u. 0x innere Schleife *)
...	...	(* es fehlen noch 7 Zeilen *)

Abstraktes Beispiel für Boundary-Interior-Test:

```
IF b1 THEN n1 ELSE n2 END;

WHILE b2 DO
   n3;
   WHILE b3 DO
      IF b4 THEN n4 ELSE n5 END;
   END;
END;
```

Durchläufe für modifizierten Boundary-Interior-Test:

n1	n2	(* kein Durchlauf *)
n1, n3	n2, n3	(* 1x äussere, 0x innere Schleife *)
n1, n3, n4	n2, n3, n4	(* 1x äussere, 1x innere Schleife *)
n1, n3, n5	n2, n3, n5	
n1, n3, n4, n4	n2, n3, n4, n4	(* 1x äussere, 2x innere Schleife *)
n1, n3, n4, n5	n2, n3, n4, n5	
n1, n3, n5, n4	n2, n3, n5, n4	
n1, n3, n5, n5	n2, n3, n5, n5	
n1, n3, n3	n2, n3, n3	(* 2x äussere, 0x innere Schleife *)

Bewertung der Kontrollflusstests:

- Anweisungsüberdeckung wird durch RTCA DO-178B-Standard für Software-Anwendungen in der Luftfahrt der **Kritikalitätsstufe C** gefordert (Software, deren Ausfall zu einer bedeutenden, aber nicht kritischen Fehlfunktion führen kann)
- Zweigüberdeckung wird durch RTCA DO-178B-Standard für Software-Anwendungen in der Luftfahrt der **Kritikalitätsstufe B** gefordert (Software, deren Ausfall zu schwerer aber noch nicht katastrophaler Systemfehlfunktion führen kann)
- modifizierte Bedingungsüberdeckung wird durch RTCA DO-178B-Standard für Software-Anwendungen in der Luftfahrt der **Kritikalitätsstufe A** gefordert (Software, deren Ausfall zu katastrophaler Systemfehlfunktion führen kann)
- Zweigüberdeckung sollte für uns Mindestanforderung beim Testen darstellen (Kontrollflussfehler/Bedingungsfehler werden relativ gut gefunden, Datenflussfehler natürlich weniger gut)
- Einfache Variante von „modified boundary interior test“ zur Ergänzung: für jede Schleife gibt es Testfälle/Pfade, die sie gar nicht, genau einmal und (mindestens) zweimal ausführen (die Randbedingung „alle Pfade“ wird komplett aufgegeben)

4.5 Datenflussbasierte Testverfahren

Ausgangspunkt ist der Datenflussgraph einer Komponente bzw. der mit Datenflussattributen annotierte Kontrollflussgraph. Bei der Auswahl von Testfällen wird darauf geachtet, dass

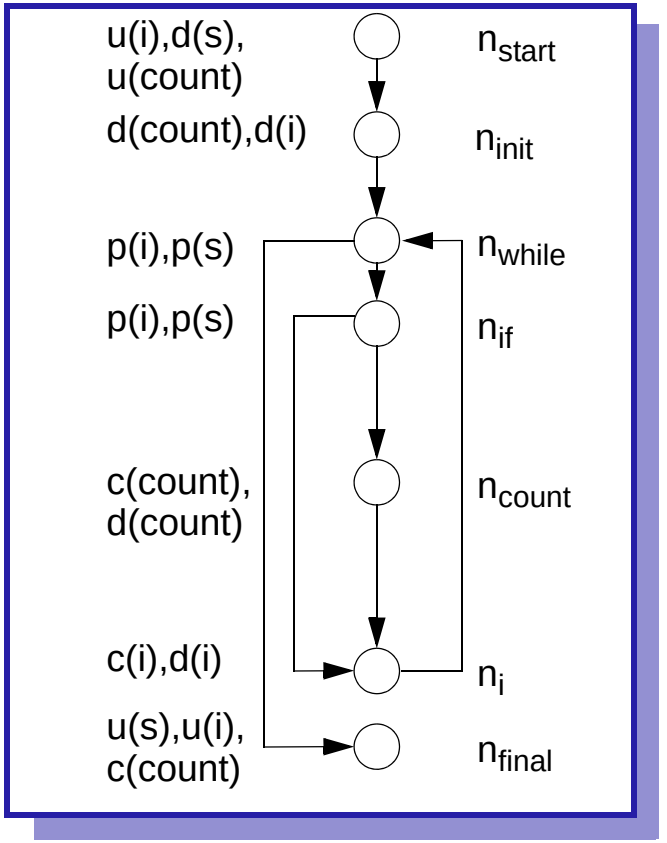
- ⇨ für jede Zuweisung eines Wertes an eine Variable **mindestens eine** (berechnende, prädikative) Benutzung dieses Wertes getestet wird
- ⇨ oder für jede Zuweisung eines Wertes an eine Variable **alle** (berechnenden, prädikativen) Benutzungen dieses Wertes getestet werden

Die datenflussbasierten Testverfahren haben folgende Vor- und Nachteile:

- ☺ einige Verfahren enthalten die Zweigüberdeckung und finden sowohl Daten-flussfehler **als auch** Kontrollflussfehler
- ☺ besser geeignet für objektorientierte Programme mit oft einfachem Kontrollfluss aber komplexem Datenfluss
- ☹ es gibt kaum Werkzeuge, die datenflussbasierte Testverfahren unterstützen

Zur Erinnerung - Kontrollflussgraph mit Datenflussattributen:

```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN                          (* start *)

   count := 0; i := 0;         (* init *)

   WHILE s[i] # '.' DO

      IF    s[i]= 'a' OR s[i]= 'e' OR
            s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

            count := count+1;

      END;

      i := i+1;

   END (* WHILE *)
END countVowels;          (* final *)
```



Kriterien für den Datenflusstest - 1:

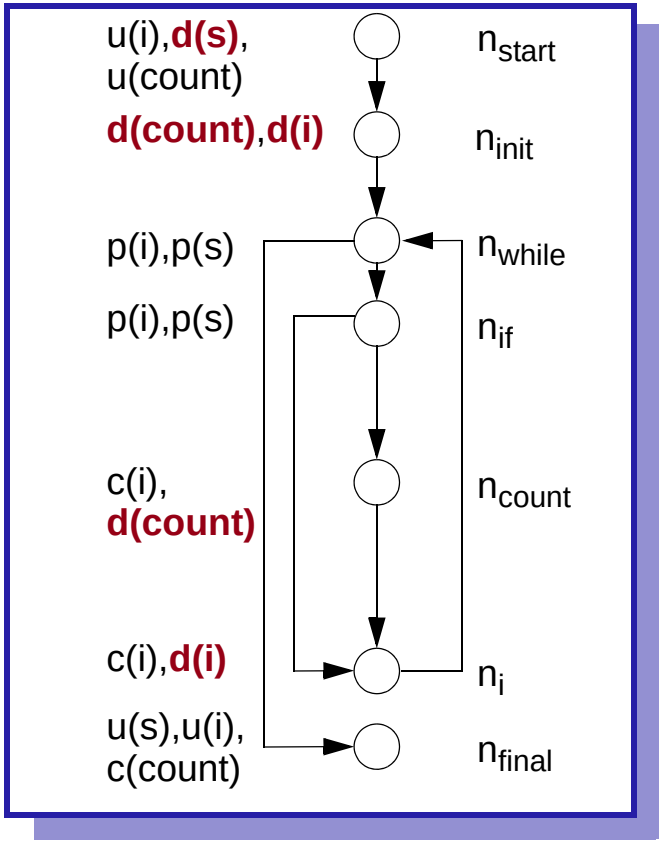
- **all-defs-Kriterium**: für jede Definitionsstelle d(x) einer Variablen muss **ein** definitionsfreier Pfad zu **einer** Benutzung r(x) existieren (und getestet werden)
  - ⇨ Kriterium kann statisch überprüft werden (entdeckt sinnlose Zuweisungen)
  - ⇨ umfasst weder Zweig- noch Anweisungsüberdeckung
  - ⇨ bei countVowels reichen Testbeispiele '.' und 'a.'
  - ⇨ Verfahren findet einige Berechnungsfehler und kaum Kontrollflussfehler
- **all-p-uses-Kriterium**: für jede Definitionsstelle d(x) wird jeweils **ein** definitionsfreier Pfade zu **allen** (erreichbaren) prädikativen Benutzungen p(x) getestet
  - ⇨ entdeckt vor allem Kontrollflussfehler
  - ⇨ Berechnungsfehler bleiben oft unentdeckt
  - ⇨ **Anmerkung**: manchmal wird auch Test **aller** definitionsfreien Pfade von d(x) zu allen p(x) gefordert, die Schleifen nicht mehrfach durchlaufen müssen (dann ist Zweigüberdeckung enthalten)

Kriterien für den Datenflusstest - 2:

- **all-c-uses-Kriterium**: für jede Definitionsstelle d(x) wird jeweils **ein** definitionsfreier Pfade zu **allen** (erreichbaren) berechnenden Benutzungen c(x) getestet
  - ⇨ entdeckt vor allem Berechnungsfehler
  - ⇨ Kontrollflussfehler bleiben oft unentdeckt
  - ⇨ lässt sich wegen Bedingungen oft nicht erzwingen
- **all-p-uses-some-c-uses-Kriterium**: für jede Definitionsstelle d(x) wird jeweils **ein** definitionsfreier Pfade zu **allen** (erreichbaren) prädikativen Benutzungen p(x) getestet; gibt es keine prädikate Benutzung p(x), so wird wenigstens **ein** Pfad zu **einem** berechnenden Zugriff c(x) betrachtet
  - ⇨ entdeckt Kontrollfluss- und auch Berechnungsfehler
  - ⇨ umfasst all-def- und all-p-uses-Kriterium
- **all-c-uses-some-p-uses-Kriterium**: … (wird kaum benutzt)
- **all-uses-Kriterium**: all-p-uses- + all-c-uses-Kriterium (wird kaum benutzt)

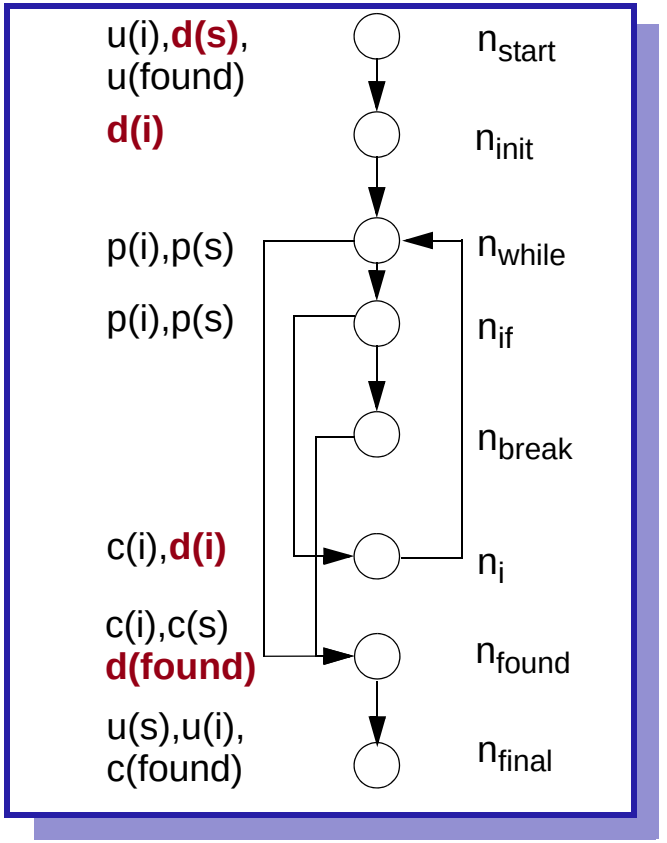
Beispiel für fehlende Zweigüberdeckung von all-defs mit '.' und 'a.':

```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN                          (* start *)

   count := 0; i := 0;         (* init *)

   WHILE s[i] # '.' DO

      IF    s[i]= 'a' OR s[i]= 'e' OR
            s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

            count := i+1;        (* Datenflussfehler *);

      END;

      i := i+1;

   END (* WHILE *)
END countVowels;          (* final *)
```



Beispiel für fehlende Anweisungsüberdeckung von all-defs mit ’b.’:

```
PROCEDURE findVowel(IN s: Sentence; OUT found: BOOLEAN);
   (* Searches for first vowel in sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;
BEGIN                          (* start *)

   i := 0;                     (* init *)

   WHILE s[i] # ‘.’ DO

      IF    s[i]= ‘a’ OR s[i]= ‘e’ OR
            s[i]= ‘i’ OR s[i]= ‘o’ OR s[i]= ‘u’
      THEN
            break; (* Schleife wird  verlassen *)
      END;

      i := i+1;

   END (* WHILE *)

   found := (s[i] # ‘.’);

END countVowels;               (* final *)
```



All-p-uses-Test am Beispiel:

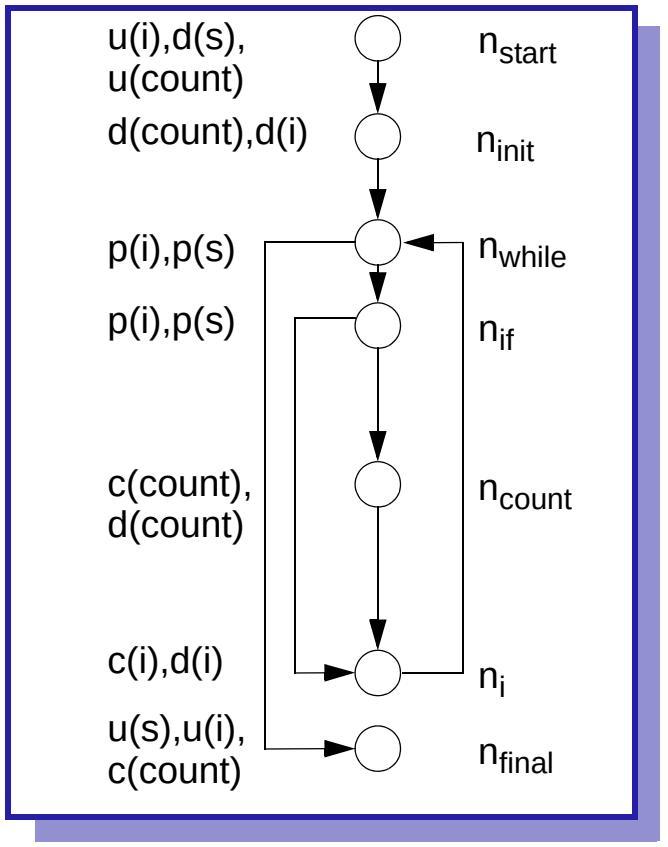
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
(Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)*

1. für d(s) bei n_{start} : Pfad zu n_{while} und n_{if}
2. für d(i) bei n_{init} : Pfad zu n_{while} und n_{if}
3. für d(i) bei n_i : Pfad zu n_{while} und n_{if}

Es reicht also jede Eingabe mit folgendem Ausführungspfad: n_{start} , n_{init} , n_{while} , n_{if} , n_i , n_{while} , n_{if} , ... , n_{while} , n_{final}

Minimaler Testfall:

'bb.'



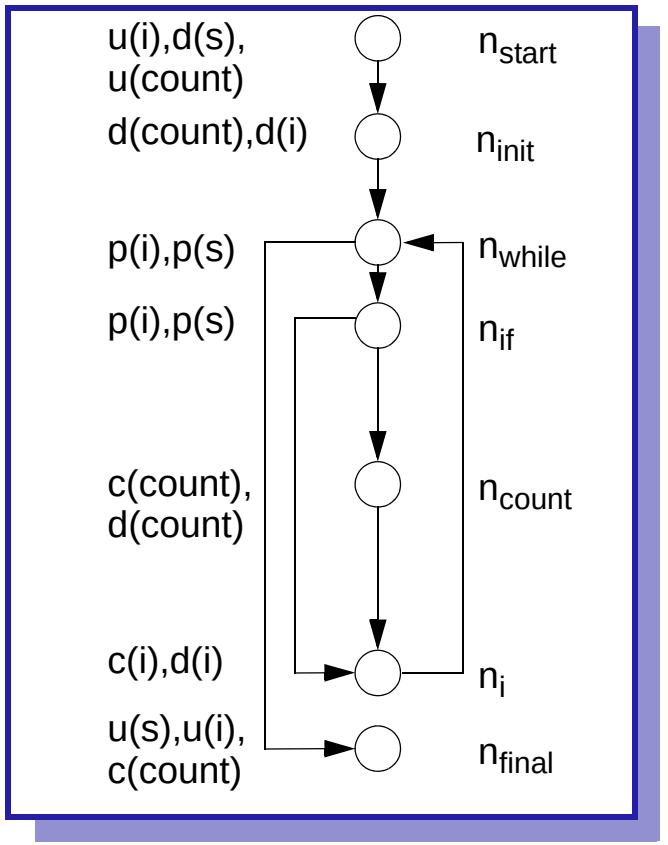
All-c-uses-Test am Beispiel:

PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
(Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)*

1. für d(i) bei n_{init} : Pfad zu n_i
2. für d(count) bei n_{init} : Pfad zu n_{count} und definitionsfreier, direkter Pfad zu n_{final}
3. für d(count) bei n_{count} : Pfad zu n_{count} und definitionsfreier, direkter Pfad zu n_{final}
4. für d(i) bei n_i : Pfad zu n_i

Minimale Anzahl von Testfällen:

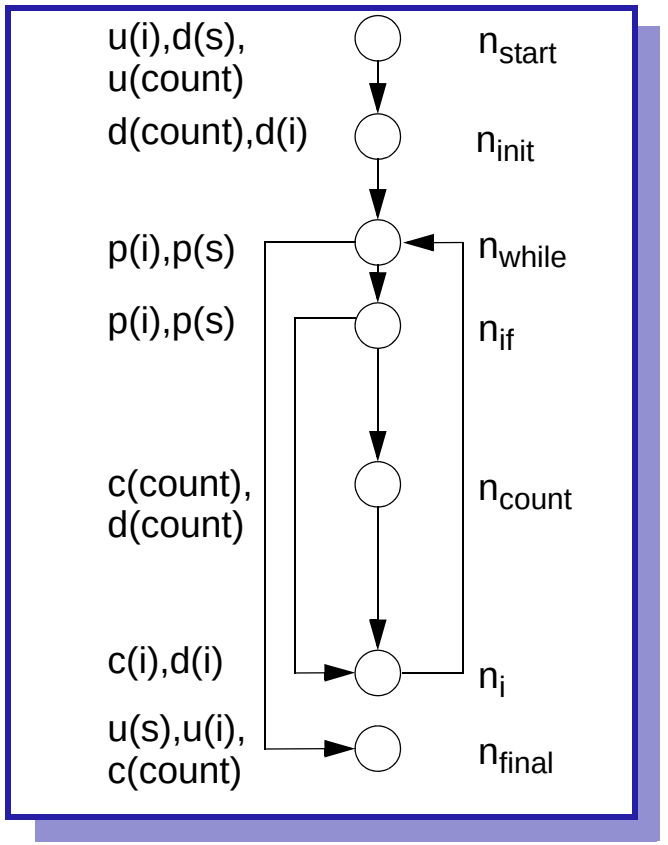
1. ’a.’
2. ’.’
3. ’aa.’
4. ’a.’



All-p-uses-some-c-uses-Test am Beispiel:

PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
(Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)*

1. für d(s) bei n_{start} : Pfad zu n_{while} und n_{if}
2. für d(i) bei n_{init} : Pfad zu n_{while} und n_{if}
3. für d(count) bei n_{init} : Pfad zu n_{count} oder definitionsfreier, direkter Pfad zu n_{final}
4. für d(count) bei n_{count} : Pfad zu n_{count} oder definitionsfreier, direkter Pfad zu n_{final}
5. für d(i) bei n_i : Pfad zu n_{while} und n_{if}



Es reicht also jede Eingabe mit folgendem Ausführungspfad: n_{start} , n_{init} , n_{while} , n_{if} , ... , n_{count} , n_i , n_{while} , n_{if} , ... , n_{while} , n_{final}

Minimaler Testfall:

'ab.'

Zusammenfassung des überdeckungsbasierten Testens:

- man wählt ein oder mehrere Überdeckungskriterien aus, die der „Kritikalität“ der zu entwickelnden Software gerecht werden
- Kombination von einem kontrollflussbasierten und einem datenflussbasierten Überdeckungskriterium sinnvoll
- man wählt nach beliebiger Methodik initiale Menge von Testfällen aus (siehe etwa Abschnitt 4.3 über funktionsorientierte Testverfahren)
- dann wird überprüft, zu wieviel Prozent die gewählten Kriterien erfüllt sind
  - ⇨ Prozentsatz ausgeführter Anweisungen beim C_0-Test
  - ⇨ Prozentsatz getesteter Definitionsstellen beim all-defs-Kriterium
  - ⇨ ...
- es werden solange Testfälle hinzugefügt, bis eine vorab festgelegte Prozentzahl für alle gewählten Überdeckungskriterien erfüllt ist (90% oder ...)
- Achtung: 100% lässt sich in vielen Fällen nicht erreichen (wegen Anomalien)

4.6 Testen objektorientierter Programme (Zustandsbezogenes Testen)

Prinzipiell lassen sich in objektorientierten Sprachen geschriebene Programme wie alle anderen Programme testen. Allerdings gibt es einige Besonderheiten, die das Testen sowohl erschweren als auch erleichtern (können):

- ☺ die Datenkapselung konzentriert Zugriffsoperationen auf Daten an einer Stelle und erleichtert damit das Testen (Einbau von Konsistenzprüfungen)
- ☹ die Datenkapselung erschwert das Schreiben von Testtreibern, die auf interne Zustände von Objekten zugreifen müssen
- ☺ die Vererbung mit Wiederverwendung bereits getesteten Codes reduziert die Menge an neu geschriebenen zu testenden Code
- ☹ die Vererbung ist eine der Hauptfehlerquellen (falsche Interaktion mit geerbten Code bzw. falsche Redefinition von geerbten Methoden)
- ☹ dynamisches Binden erschwert die Definition sinnvoller Überdeckungsmetriken ungemein (beim White-Box-Test)
- ☹ Verhalten von Objektmethoden ist (fast) immer zustandsabhängig

Prinzipien des Tests objektorientierter Programme:

- beim **„Black-Box“-Test** wie bisher vorgehen (Objekte als Eingabeparameter werden gemäß interner Zustände verschiedenen Äquivalenzklassen zugeordnet)
- beim **„White-Box“-Test** werden Kontrollflussgraphen erweitert, um so Effekte des dynamischen Bindens mit zu berücksichtigen (wird hier nicht weiter verfolgt)
- **Zustandsautomaten** werden zusätzlich zu Kontrollflussgraphen zur Testplanung herangezogen (dieses Verfahren wird oft dem „Black-Box“-Test zugeordnet)
- Einbau von **Konsistenzüberprüfungen** (Plausibilitätsüberprüfungen, assert in Java), in Methoden (zu Beginn und nach Abarbeitung von Methodencode)
- [**defensive Programmierung**: Code fängt alle erkennbaren Inkonsistenzen ab]
- **geerbter Code** wird wie neu geschriebener Code behandelt und immer vollständig im Kontext der erbenden Klasse neu getestet
- inkrementelles Testen mit **Regressionstests** unter Verwendung von Frameworks wie JUnit (http://www.junit.org); siehe Softwarepraktikum

Prinzipien beim Test einzelner Klassen - 1:

Es werden in [Bi00] vier verschiedene Arten von Klassen unterschieden:

1. **Nicht-modale Klassen**: Methoden der Klasse können immer (zu beliebigen Zeitpunkten) aufgerufen werden; interner Zustand der Objekte spielt dabei keine Rolle
   - ⇨ Methoden können isoliert für sich getestet werden; bei der Auswahl der Testfälle muss Objektzustand nicht mit berücksichtigt werden
   - ⇨ Beispiel: Gerätesteuerung mit setStatus-Methode u. getStatus-Methode, die Zustand liefert (Beispiel ist Grenzfall; besser Klasse ohne Attribute)

2. **Uni-modale Klasse**: Methoden können nur - unabhängig vom internen Zustand der Objekte - in einer bestimmten Reihenfolge aufgerufen werden (warum?)
   - ⇨ Testfälle müssen alle zulässigen und nicht zulässigen Reihenfolgen von Methodenaufrufen durchprobieren; interne Objektzustände nicht relevant (Automaten mit zulässigen Methodenaufrufen als Transitionen werden zur Testplanung herangezogen)
   - ⇨ Beispiel: Gerätesteuerung mit init-, setStatus- und getStatus-Methoden; init-Methode muss zuerst aufgerufen werden

Prinzipien beim Test einzelner Klassen - 2:

3. **Quasi-modale Klasse**: Zustand der Objekte bestimmt zulässige Reihenfolge von Methodenaufrufen (und nur dieser)
   - ⇨ Methoden werden isoliert getestet, aber für alle zu unterscheidenden Äquivalenzklassen des internen Objektzustandes (Automaten mit Objekt-zuständen werden zur Testplanung herangezogen)
   - ⇨ Beispiel: Gerätesteuerung mit setStatus-Methode und getStatus-Methode, die nur 100.000 Status-Wechsel zulässt und dann den Dienst verweigert (nach Wartung verlangt)

4. **Modale Klasse**: Methoden können nur in fest vorgegebenen Reihenfolgen aufgerufen werden; zusätzlich hat Objektzustand Einfluss auf Zulässigkeit von Aufrufen
   - ⇨ Kombination der Testmethoden für uni-modale und quasi-modale Klassen notwendig; Testplanung mit Automaten
   - ⇨ Beispiel: Gerätesteuerung mit init-, setStatus- und getStatus-Methode mit zusätzlicher Beschränkung auf 100.000 Status-Wechsel

© Prof. Dr. Andy Schürr (TU Darmstadt, FB 18, Institut für Datentechnik)

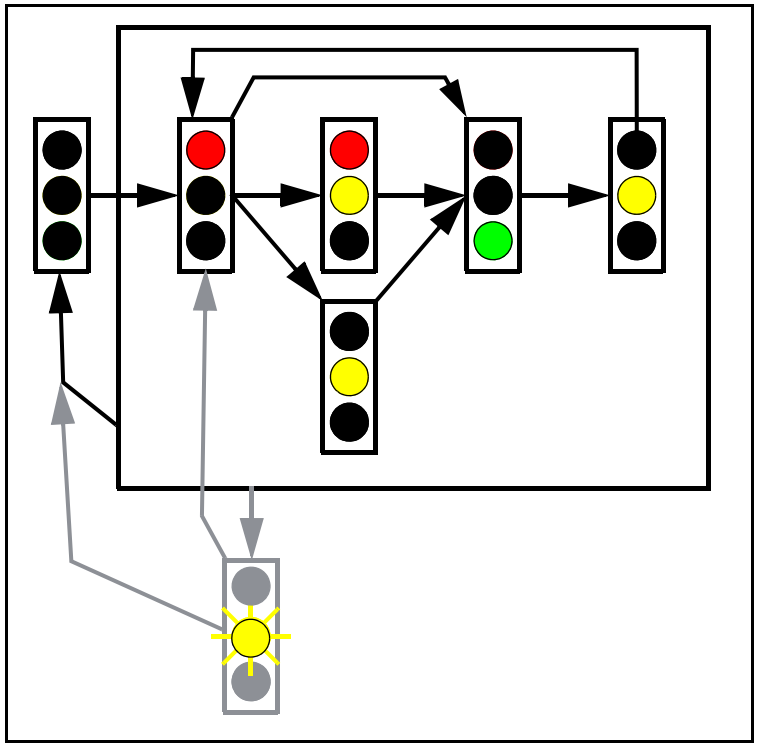
Tabellarische Übersicht über verschiedene Arten von Klassen beim Testen:

	Zustand bestimmt nicht Methodenaufrufbarkeit	**Zustand bestimmt Methodenaufrufbarkeit**
Aufrufreihenfolge der Methoden flexibel	**nicht modal**: Methoden isoliert testen	**quasi-modal**: Methoden isoliert testen für alle Zustandsäquivalenz-klassen
Aufrufreihenfolge der Methoden fest	**uni-modal**: alle zulässigen und verbote-nen Reihenfolgen testen	**modal**: alle zulässigen/verbotenen Reihenfolgen testen für alle Zustandsäquivalenzklassen

Beispiel Ampelsteuerung - modale Klasse:

Eine Klasse Ampel (TrafficLight) wird im folgenden als Beispiel verwendet. Sie bietet Grundfunktionen für die Realisierung von Ampelsteuerungen an und „verkapselt“ die Gemeinsamkeiten der Ampelsteuerungen verschiedener Länder:

- eine Ampel ist aus oder im Rot-Gelb-Grün-Zyklus oder blinkt (Zusatzfunktion)
- von Rotphase wird entweder über Gelb nach Grün gewechselt oder direkt
- in Gelbphase nach Rot leuchtet entweder nur gelbes Licht oder auch rotes Licht
- eine Ampel kann immer ausgeschaltet werden oder nach Blinken wechseln
- vom Zustand Blinken wechselt eine Ampel immer nach Rot
- eine defekte Ampel (ein Licht geht nicht) schaltet sich automatisch ab



Prinzipien bei der OO-Implementierung der Ampelsteuerung:

- folgende Arten von Methoden werden unterschieden (und später beim Testen unterschiedlich behandelt):
  - ⇨ **Konstruktoren** und **Destruktoren**, die Objekt erzeugen bzw. löschen
  - ⇨ beobachtende Methoden (**Observer**), die Objektzustand nicht ändern
  - ⇨ verändernde Methoden (**Modifier**)
- es werden sogenannte **Invarianten** (Invariants) aufgeschriebenen, die nach bzw. vor Ausführung aller Methoden immer gelten müssen:
  - ⇨ Invarianten können durch Observer nicht zerstört werden
  - ⇨ Invarianten müssen am Ende des Codes eines Konstruktors gelten
  - ⇨ Invarianten gelten vor Aufruf eines Modifiers und müssen am Ende des Codes eines Modifiers wieder gelten
- für Observer, Modifier und Destruktoren können **Vorbedingungen** (Preconditions) angegeben werden, die bei Aufruf (zusätzlich zu Invarianten) gelten
- für Konstruktoren, Modifier (und Observer) können **Nachbedingungen** (Postconditions) angegeben werden, die nach Ausführung zusätzlich zu Invarianten gelten

Implementierung der Ampelsteuerung in C++:

```
class TrafficLight {

public:
// invariant: (isOff() || isGreen() || isYellow() || isRed());
// invariant: (! ( isOff() && (isGreen() || isYellow() || isRed()) ) );

// Konstruktoren und Destruktoren
   TrafficLight();
   ~TrafficLight();

// öffentliche beobachtende Operationen (Observer)
   bool isGreen() const;
   bool isYellow() const;
   bool isRed() const;
   bool isOff() const;

// öffentliche verändernde Operationen (Modifier)
   virtual void lightOn();
      // precondition: isOff();
      // postcondition: isRed();
   virtual void lightOff();
      // precondition: (isGreen() || isYellow() || isRed());
      // postcondition: isOff();
```

Implementierung der Ampelsteuerung - Fortsetzung:

```
    virtual void offOnFault();
      // precondition: (! isOff());
      // postcondition: ( (greenOk() && yellowOk() && redOk()) || isOff() )

    virtual void greenOn();
      // precondition: isYellow() || isRed();
      // postcondition: isGreen();
    virtual void yellowOn();
      // precondition: isGreen() || isRed();
      // postcondition: isYellow();
    virtual void redOn();
      // precondition: isYellow();
      // postcondition: isRed();

private:
    bool off, green, yellow, red;
    // Initialisierung: off = true, green = yellow = red = false;

protected:
    // Überprüfung der Funktionsfähigkeit der drei Lampen
    bool greenOk(); bool yellowOk(); bool redOk();

};
```

Prinzipien bei der Spezialisierung der Ampelsteuerung:

- es dürfen beliebig neue Methoden, Attribute, ... als neue Funktionalität hinzugefügt werden
- es dürfen neue Invarianten hinzugefügt werden bzw. die bestehenden Invarianten der Klasse verschärft werden (geerbte und neue Methoden erhalten mindestens die geerbten Invarianten)
- es dürfen Methoden redefiniert werden, wenn dabei
  - ⇨ Vorbedingungen allenfalls gelockert werden (redefinierte Methode ist mindestens dann aufrufbar, wenn geerbte Vorbedingungen erfüllt sind)
  - ⇨ Nachbedingungen allenfalls verschärft werden (redefinierte Methode erfüllt mindestens die geerbten Nachbedingungen)

Achtung:

Werden Invarianten verschärft (hinzugefügt), so müssen auch die unverändert gebliebenen geerbten Methoden diese neuen Invarianten erfüllen!

Implementierung der Spezialisierung in C++:

```
class FlashingTrafficLight : public TrafficLight {
   // bietet neue Operation „blinkendes gelbes Licht“ an
   // verbietet direkten Übergang von rotem Licht zu grünem Licht
   // bei der Gelbphase nach der Rotphase ist auch das rote Licht an

public:

// erlaubte Verschärfung der geerbten Invarianten:
// invariant: (! ( (isOff() || isFlashing()) && (isGreen() || isYellow() || isRed()) ) );
// invariant: (! isGreen() && (isRed() || isYellow()) );
// invariant: (! (isOff() && isFlashing());

// verbotene Lockerung der geerbten Invarianten:
// invariant: (isOff() || isFlashing() || isGreen() || isYellow() || isRed())

// neue Observer-Operation
   bool isFlashing();

// neue Modifier-Operation
   virtual void flashingOn();
      // precondition: (isRed() || isYellow() || isGreen());
      // postcondition: isFlashing();
```

Erlaubte Verschärfung geänderter Invarianten:

```
class FlashingTrafficLight : public TrafficLight {
   // bietet neue Operation „blinkendes gelbes Licht“ an
   // verbietet direkten Übergang von rotem Licht zu grünem Licht
   // bei der Gelbphase nach der Rotphase ist auch das rote Licht an

public:

// invariant: (isOff() || isOn());  geänderte geerbte Bedingung v. TrafficLight
// invariant: !(isGreen() || isYellow() || isRed())  || isOn();  neue geerbte Bedingung
// ...

// erlaubte Verschärfung der geerbten Invarianten:
// invariant: !isFlashing() || isOn();   hinzugefügte Bedingung in FlashingTrafficLight

// neue Observer-Operation
   bool isFlashing();

// neue Modifier-Operation
   virtual void flashingOn();
      // precondition: (isRed() || isYellow() || isGreen());
      // postcondition: isFlashing();
```

Implementierung der Spezialisierung - Fortsetzung:

```
// Redefinitionen
   virtual void greenOn();
      // precondition: isYellow();  verbotene Verschärfung von (isYellow() || isRed())
      // postcondition: isGreen();
   virtual void yellowOn();
      // precondition: isGreen() || isRed();
      // postcondition: isYellow() && isRed(); erlaubte Verschärfung der Nachbedingung
   virtual void redOn();
      // precondition: isYellow() || isFlashing(); erlaubte Lockerung der Vorbedingung
      // postcondition: isRed();

private:
   bool flashing;

};
```

Anmerkung:

Die Einschränkung, dass die „blinkende Ampel“ nur noch in „grün“ schalten darf, wenn sie im Zustand „gelb“ ist, führt dazu, dass sich die blinkende Ampel nicht mehr wie eine „normale“ Ampel der Klasse TrafficLight verhält.

Vorüberlegungen zur Realisierung von Invarianten, ... :

- die teuere Überprüfung von Invarianten, ... muss durch „Compile“-Flag abschaltbar sein (entweder systemweit oder je Subsystem)
- die Überprüfungen dürfen das Verhalten des ausführbaren Programms (abgesehen von Laufzeit und Speicherplatzverbrauch) nicht verändern
- bei abgeschalteten Überprüfungen sollte der Code für diese Überprüfungen nicht Bestandteil des ausführbaren Programms sein
- die Reaktion auf fehlgeschlagene Überprüfungen muss an einer Stelle (veränderbar) festgelegt werden (Programmabbruch, Ausnahmeerweckung, ...)
- die Überprüfungen sollten möglichst lesbar niedergeschrieben werden
- Invarianten werden auch vor der Ausführung einer Methode und nach der Ausführung von „Observer“-Methoden überprüft (um illegale Objektzugriffe entdecken zu können)

Zusicherungen (Assertions) in Java:

- Java besitzt ab Version 1.4 „assert“-Statement, das für den Einbau von Überprüfungen (Vor-/Nachbedingungen, Invarianten) genutzt wird
- die Überprüfung von „assert“-Statements kann durch Runtime-Flags generell oder klassenweise an- bzw. abgeschaltet werden („-enableassertions“ = „-ea“ und „.-disableassertions“ = „-da“)
- die Überprüfungen sollten das Verhalten des ausführbaren Programms (abgesehen von Laufzeit und Speicherplatzverbrauch) nicht verändern (der Programmierer muss das sicherstellen)
- soll der Code für „assert“-Statements nicht Bestandteil des ausführbaren Programms sein, so muss der Compiler davon „überzeugt“ werden, dass der Code wegoptimiert werden kann:

```
static final boolean assertChecksOn = ... ; // false to eliminate asserts
if (assertChecksOn) assert ... ;
```

- „assert“-Verletzungen können als „AssertionError“-Ausnahmen abgefangen werden (und damit im Code festgelegte Reaktionen auslösen)

Makros für die Überprüfung von Invarianten, ... in C++:

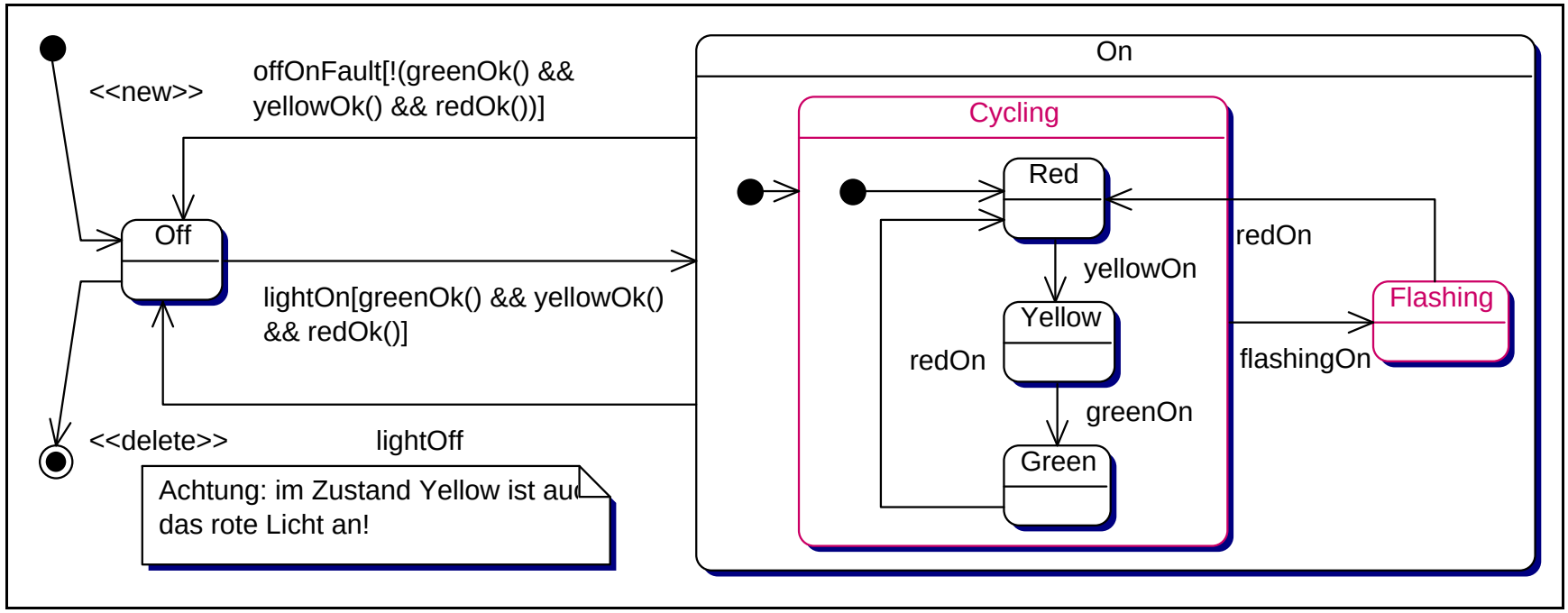
```
# ifdef DEBUG
      # define ASSERT(condition) {                                    \
            if (!(condition)) {                                       \
               < raise exception or terminate program or print error message or ... >   \
            }                                                         \
      }
      // nur im Debug-Modus werden Zusicherungen überprüft
# else
   # define ASSERT(condition)
# endif

# define INVARIANT  ASSERT( invariant() )
// im Debug-Modus wird als „protected“ Methode jeder Klasse codierte Invariante überprüft

# define POST(condition) {                                          \
      ASSERT(condition); INVARIANT;                                 \
   }
// zusätzlich zu Nachbedingung wird Einhaltung der Invariante sichergestellt

# define PRE(condition) {                                           \
      INVARIANT; ASSERT(condition);                                 \
   }
```

Überprüfung von Invarianten, ... in C++:

```
class TrafficLight {
public:
// Konstruktoren und Destruktoren
   TrafficLight() {
      ... ; INVARIANT;
   };

   ~TrafficLight() {
      INVARIANT; ...
   };

// öffentliche verändernde Operationen (Modifier)
   void lightOn() {
      PRE(isOff());
      ... ;
      POST(isRed());
   };

 ...

protected:
   inline bool invariant() const { return ( ... ); };
   // Übersetzer ruft inline-Methode nicht auf, sondern kopiert Rumpf an Aufrufstelle ein
};
```

Einsatz von Automaten (Statecharts) zur Testplanung:

Oft lassen sich die Vorbedingungen für den Aufruf von Methoden besser durch ein Statechart (hierarchischer Automat, siehe Software Eng. - Einführung) darstellen:



Ein solches Statechart kann dann - ähnlich wie ein Kontrollflussgraph - zur Planung von Testfällen zur Berechnung von Testüberdeckungsmetriken herangezogen werden.

Präzisierung der vier verschiedenen Arten von Klassen:

- Nicht modale Klasse:
  - ⇨ keine Methode der Klasse hat eine Vorbedingung
  - ⇨ der Zustandsautomat der Klasse besteht aus einem Zustand
- Quasimodale Klasse:
  - ⇨ mindestens eine Methode der Klasse hat eine Vorbedingung
  - ⇨ der Zustandsautomat der Klasse besteht aus einem Zustand
- Unimodale Klasse:
  - ⇨ der Zustandsautomat der Klasse besteht aus mehreren Zuständen
  - ⇨ keine Transition des Automaten hat eine Bedingung / Wächter (zusätzlich zu den Zuständen gibt es keine Vorbedingungen an Methoden)
- Modale Klasse:
  - ⇨ der Zustandsautomat der Klasse besteht aus mehreren Zuständen
  - ⇨ mindestens eine Transition des Automaten hat eine Bedingung / Wächter (zusätzlich zu den Zuständen gibt es Vorbedingungen an Methoden)

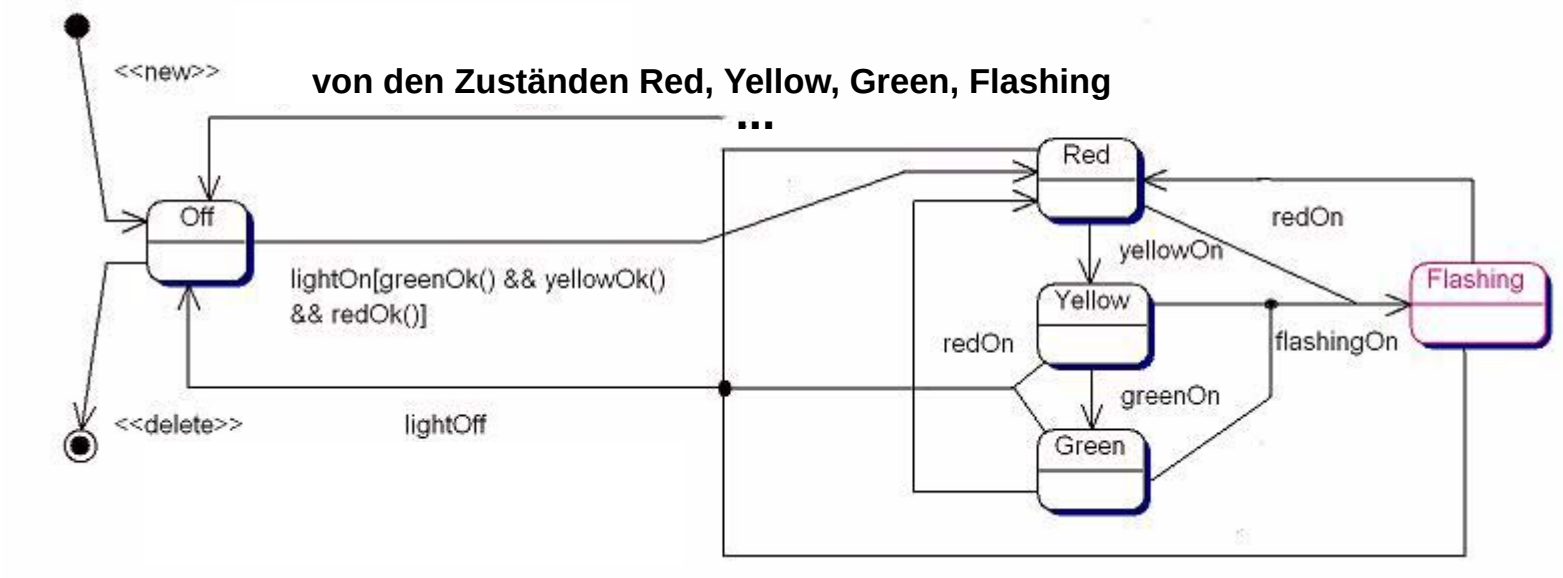
Definition von Testüberdeckungsmetriken für Statecharts:

- Tests müssen garantieren, dass alle Zustände mindestens einmal erreicht werden (entspricht Anweisungsüberdeckung aus Abschnitt 4.4)
- Tests müssen garantieren, dass jede Transition mindestens einmal ausgeführt wird (entspricht Zweigüberdeckung aus Abschnitt 4.4), nachdem hierarchisches Statechart in flachen Automaten übersetzt wurde (siehe SW Eng. - Einführung)
- Tests müssen garantieren, dass jede Transition mit allen sich wesentlich unterscheidenden Belegungen ihrer Bedingung ausgeführt wird (entspricht modifiziertem Bedingungsüberdeckungstest aus Abschnitt 4.4)
- Tests müssen alle möglichen Pfade durch Statechart (bis zu einer vorgegebenen Länge oder vorgegebenen Anzahl von Zyklen) ausführen (entspricht eingeschränkten Varianten der Pfadüberdeckung aus Abschnitt 4.4)
- zusätzlich zum Test aller explizit aufgeführten Transitionen werden für jeden Zustand alle sonst möglichen Ereignisse (Methodenaufrufe) ausgeführt (gewünschte Reaktion: Ignorieren oder Programmabbruch oder ...)

Testplanung mit Transitionsbaum (Übergangsbaum) aus [Bi00]:

1. das gegebene Statechart wird in einen flachen Automaten übersetzt
2. Transitionen mit komplexen Boole'schen Bedingungen werden in mehrere Transitionen mit Konjunktion atomarer Bedingungen übersetzt
   (Transition mit [(a1 && a2) || (b1 && b2)] wird ersetzt durch Transition mit [a1 && a2] und Transition mit [b1 && b2]
3. ein Baum wird erzeugt, der
   - ⇨ initialen Zustand als Wurzelknoten (ersten, obersten Knoten) besitzt
   - ⇨ Zustandsknoten im Baum werden expandiert, indem alle Transitionen zu anderen Zuständen (und sich selbst) als Kindknoten hinzugefügt werden
   - ⇨ jeder Zustand wird nur einmal als Knoten im Transitionsbaum expandiert
4. jeder Pfad in dem Baum (von Wurzel zu einem Blatt) entspricht einer Testsequenz
5. zusätzlich werden in jedem Zustand alle Ereignisse ausgelöst, die nicht im Transitionsbaum aufgeführt sind (spezifikationsverletzende Transitionen)

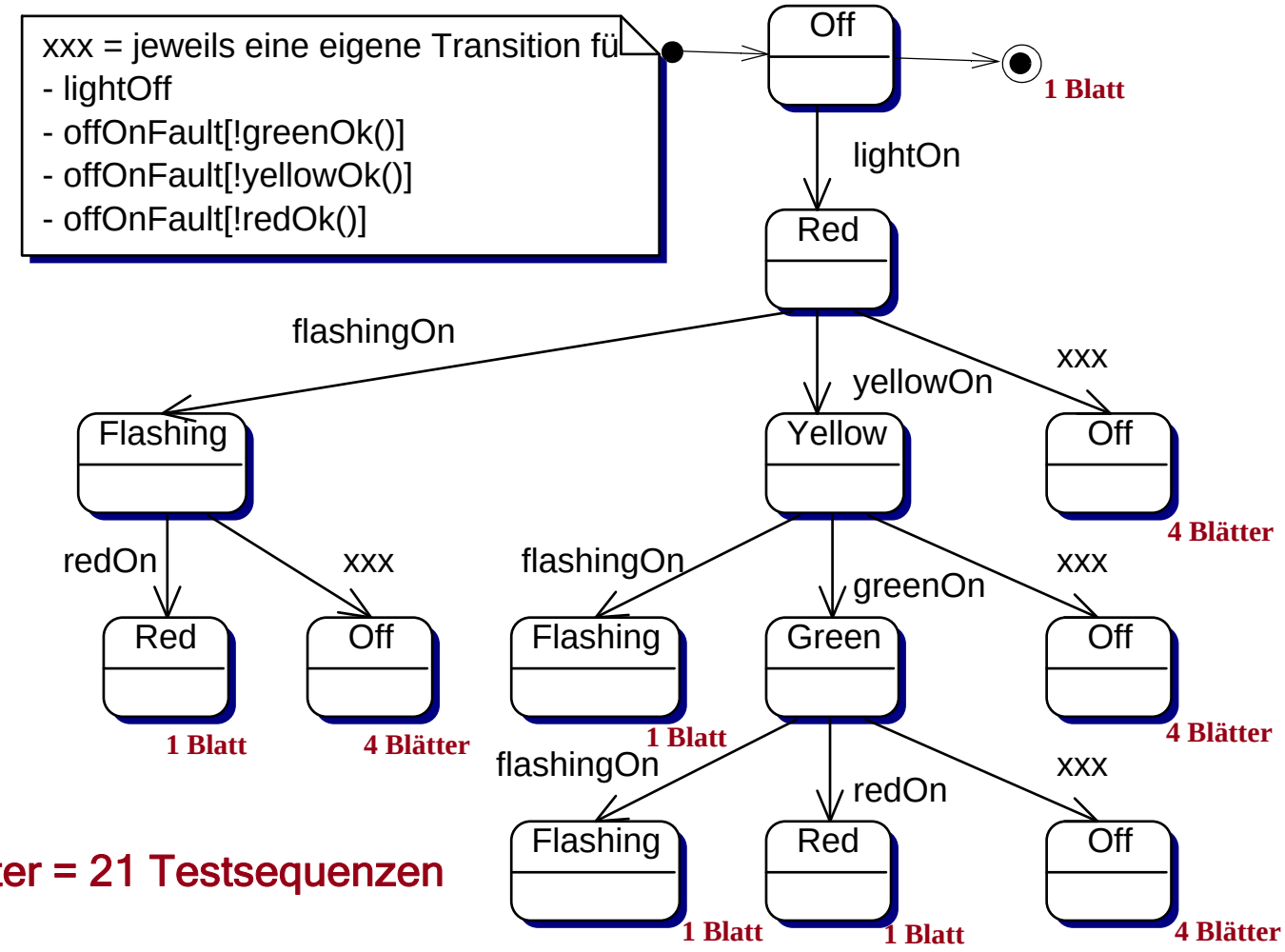
„Flachgeklopfter“ Automat mit einfacheren Transitionsbedingungen:



In dem flachen Automaten (Statechart) gibt es zusätzlich zu den eingezeichneten Transitionen von jedem der Zustände Red, Yellow, Green, Flashing jeweils drei Transitionen zu dem Zustand Off mit den Aufschriften

- ⇨ offOnFault [! greenOk()]
- ⇨ offOnFault [! yellowOk()]
- ⇨ offOnFault [! redOk()]

Ein möglicher Transitionsbaum für Ampelsteuerung:



21 Blätter = 21 Testsequenzen

Die normalen Testsequenzen:

1. new (Konstruktoraufruf), delete (Destruktoraufruf)
2. new, lightOn[greenOk() & yellowOk() & redOk()], flashingOn, redOn
3. new, lightOn[...], flashingOn, lightOff
4. new, lightOn[...], flashingOn, offOnFault[!greenOk()]
5. new, lightOn[...], flashingOn, offOnFault[!yellowOk()]
6. new, lightOn[...], flashingOn, offOnFault[!redOk()]
7. new, lightOn[...], yellowOn, flashingOn
8. new, lightOn[...], yellowOn, greenOn, flashingOn
9. new, lightOn[...], yellowOn, greenOn, redOn
10. new, lightOn[...], yellowOn, greenOn, lightOff
11. new, lightOn[...], yellowOn, greenOn, offOnFault[!greenOk()]
12. new, lightOn[...], yellowOn, greenOn, offOnFault[!yellowOk()]
13. new, lightOn[...], yellowOn, greenOn, offOnFault[!redOk()]
14. new, lightOn[...], yellowOn, lightOff
15. ...

Die Tests für irrelevante/irreguläre Transitionen:

1. new, lightOff — Test für den Zustand Off
2. new, lightOn[!greenOk()]
3. new, lightOn[!yellowOk()]
4. new, lightOn[!redOk()]
5. new, offOnFault[...]
6. new, redOn
7. new, yellowOn
8. new, greenOn
9. new, flashingOn
10. new, lightOn, FlashingOn, lightOn — Test für den Zustand Flashing
11. ...

Behandlung ereignisloser Transitionen:

Automatenmodelle für die Verhaltensbeschreibung wie UML-Statecharts erlauben oft auch die Definition ereignisloser Transitionen. Diese werden wie folgt beim Testen behandelt:

- Transition **ohne Ereignis mit Bedingung** [b]: sobald die Bedingung erfüllt ist, schaltet die Transition; beim Aufstellen von Testsequenzen sind zwei Fälle zu unterscheiden (zwei Unterbäume im Transitionsbaum):
  - ⇨ Bedingung b ist bereits erfüllt, wenn Startzustand der Transition betreten wird (Transition wird mit der Bedingung „[b == true]“ markiert und schaltet sofort bei der Testausführung)
  - ⇨ Bedingung ist nicht erfüllt, wenn Startzustand betreten wird; wird später aber erfüllt (Transition wird mit Ereignis „b -> true“ markiert und schaltet sobald die Bedingung b erfüllt ist)
- Transition **ohne Ereignis und ohne Bedingung**: die Transition schaltet, sobald der Startzustand betreten wird: der Startzustand erhält im Transitionsbaum eine ausgehende Kante/Transition ohne Markierung

Ausführung der Testsequenzen:

- **Testvorbereitung:** Objekt muss in initialen Zustand (zurück-)versetzt werden
- **Testausführung** einer Sequenz von Methodenaufrufen (Testvektor):
  es wird unterschieden
  - ⇨ **white-box-Sicht**: es gibt Zugriffsoperationen für Abfrage des internen Zustands; man kann also am Ende einer Testsequenz abfragen, ob richtiger Zustand erreicht wurde (interne Zustände der Implementierung müssen mit „extern“ definierten Zuständen korrespondieren)
  - ⇨ **black-box-Sicht**: Aussenverhalten muss überprüft werden - im Beispiel der Ampel „beobachtbares Verhalten“ der Ampel + ggf. Test, ob bei weiteren Eingaben sich die Steuerung so verhält, wie sie sich im entsprechenden Zustand verhalten müsste
- **Testbeendigung**: ggf. wird Sequenz von Methodenaufrufen so vervollständigt, dass am Ende der Ausführung Objekt sich in einem „terminalen“ Zustand befindet
- **Kombination** von Testsequenzen: um Aufwand für Initialisierung zu reduzieren, werden möglichst lange Testsequenzen generiert bzw. kombiniert

© Prof. Dr. Andy Schürr (TU Darmstadt, FB 18, Institut für Datentechnik)

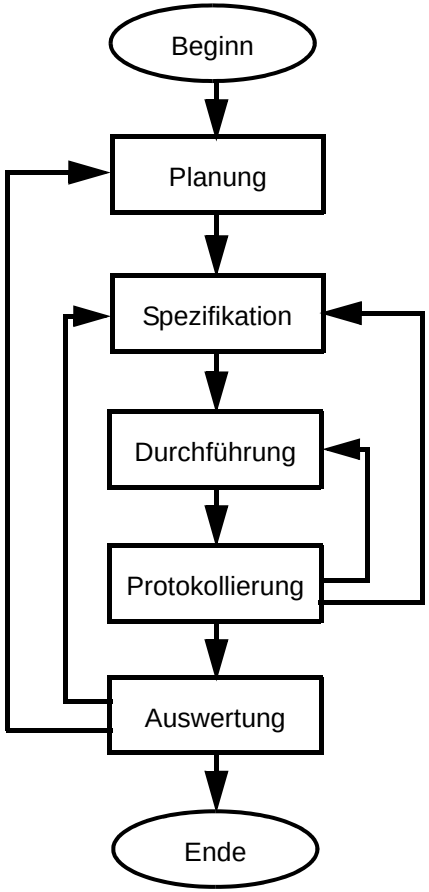
Abschließende Checkliste für Klassentest nach [Bi00]:

- jede Methode (auch die geerbten) wird mindestens einmal ausgeführt
- alle Methodenparameter und alle nach aussen sichtbaren Attribute werden mit geeigneter Äquivalenzklassenbildung durchgetestet
- alle auslösbaren (ausgehenden) Ausnahmen werden mindestens einmal ausgelöst
- alle von gerufenen Methoden auslösbaren (eingehenden) Ausnahmen werden mindestens einmal behandelt (oder durchgereicht)
- alle identifizierten Objektzustände (auch hier Äquivalenzklassenbildung) werden beim Testen erreicht
- jede zustandsabhängige Methode wird in jedem Zustand ausgeführt (auch in den Zuständen, in denen ihr Aufruf nicht zulässig ist)
- alle möglichen Zustandsübergänge (mit allen Kombinationen von Bedingungen an den Übergängen) werden aktiviert
- zusätzlich werden die üblichen Performanz-, Stress-, ... -Tests durchgeführt

4.7 Testmanagement und Testwerkzeuge

Testen wird nach [SL12] immer wie durchgeführt:

- **Testplanung**: es werden die zum Einsatz kommenden Methoden und Werkzeuge festgelegt
- **Testspezifikation**: die Testfälle werden entweder manuell oder durch Werkzeuge generiert festgelegt
- [Testimplementierung: die festgelegten Testfälle werden (automatisch ausführbar) implementiert]
- **Testdurchführung**: die ausgewählten Testfälle werden (durch Werkzeuge automatisiert) ausgeführt
- **Testprotokollierung**: Testergebnisse werden protokolliert sowie erreichte Testüberdeckung, ggf. Auswahl weiterer Testfälle notwendig
- **Testauswertung**: Bericht über Reifegrad des Testobjektes, Testaufwand, erreichter Überdeckungsgrad, ggf. weiterer Testzyklus notwendig



Aufgaben, Qualifikationen und Rollen nach [SL12]:

- **Testmanager** (Leiter): ist für Testplanung sowie Auswahl von Testwerkzeugen zuständig und vertritt „Testinteressen“ gegenüber Projektmanagern
- **Testdesigner** (Analyst): erstellt Testspezifikationen und ermittelt Testdaten
  [Ergänzung: könnte beim modellbasierten Testen auch für die Erstellung von Testmodellen und Festlegung von Überdeckungskriterien zuständig sein]
- [**Testfallgenerator**: werden Testfälle aus Modellen bzw. Spezifikationen automatisch erzeugt, so bedarf es einer weiteren Rolle, die die Testfallgenerierung mit Werkzeugunterstützung durchführt]
- **Testautomatisierer**: realisiert die automatisierte Durchführung der spezifizierten Testfälle durch ausgewählte Testwerkzeuge
- **Testadministrator**: stellt die Testumgebung mit ausgewählten Testwerkzeugen zur Verfügung (zusammen Systemadministratoren etc.)
- **Tester**: ist für Testdurchführung, -protokollkierung und -auswertung zuständig (entspricht „Certified Tester (Foundation Level“-Kompetenzen)

Weitere Aspekte des Testmanagements nach [SL12]:

- Betrachtung von **Kosten- und Wirtschaftlichkeitsaspekten**
  - ⇨ Ermittlung und Abschätzung von Fehlerkosten
  - ⇨ Ermittlung und Abschätzung von Testkosten / Testaufwand
- Wahl einer **Teststrategie**
  - ⇨ proaktiv vs. reaktiv (Testmanagement startet mit Projektbeginn vs. Testaktivitäten werden erst nach der Erstellung der Software gestartet)
  - ⇨ Testerstellungsansatz / Überdeckungskriterien / ...
  - ⇨ Orientierung an Standards für Vorgehensmodelle (vgl. Kapitel 5)
- Testen und **Risiko**
  - ⇨ Risiken beim Testen (Ausfall von Personal, ...)
  - ⇨ Risikobasiertes Testen (Fokus auf Minimierung von Produktrisiken)

Meldung von Fehlern / Fehlermanagement:

Es müssen alle Informationen erfasst werden, die für das Reproduzieren eines Fehlers notwendig sind. Eine Fehlermeldung kann wie folgt aufgebaut sein:

- **Status**: Bearbeitungsfortschritt der Meldung (Neu, Offen, Analyse, Abgewiesen, Korrektur, Test, Erledigt)
- **Klasse**: Klassifzierung der Schwere des Problems (Menschenleben in Gefahr, Systemabsturz mit Datenverlust, ... , Schönheitsfehler)
- **Priorität**: Festlegung der Dringlichkeit, mit der Fehler behoben werden muss (sofort, da Arbeitsablauf blockiert beim Anwender, ...)
- **Anforderung**: die Stelle(n) in der Anforderungsspezifikation, auf die sich der Fehler bezieht
- **Fehlerquelle**: in welcher Softwareentwicklungsphase wurde der Fehler begangen
- **Fehlerart**: Berechnungsfehler, ...
- **Testfall**: genaue Beschreibung des Testfalls, der Fehler auslöst
- ...

Arten von Testwerkzeugen - 1:

Testwerkzeuge werden oft „**Computer-Aided Software Test**“-Werkzeuge genannt (**CAST-Tools**). Man unterscheidet folgene Arten von CAST-Tools:

- Werkzeuge zum **Testmanagement** und zur Testplanung: Erfassen von Testfällen, Abgleich von Testfällen mit Anforderungen, Verwaltung und (statistische) Auswertung von Fehlermeldungen, ...
- Werkzeuge zur **Testspezifikation**: Testdaten und Soll-Werte für zugehörige Ergebnisse werden (manuell) festgelegt und verwaltet
- **Testdatengeneratoren**: aus Softwaremodellen, Code, Grammatiken, ... werden automatisch Testdaten generiert
- **Testtreiber**: passend zu Schnittstellen von Testobjekten werden Testtreiber bzw. Testrahmen zur Verfügung gestellt, die die Aufruf von Tests mit Übergabe von Eingabewerten, Auswertung der Ergebnisse etc. abwickeln
- **Simulatoren**: bilden möglichst realitätsnah Produktionsumgebung nach (mit Simulation von anderen Systemen, Hardware, ...)

Arten von Testwerkzeugen - 2:

- **Testroboter** (Capture & Replay-Werkzeug): zeichnet interaktive Benutzung einer Bedienungsoberfläche auf und kann Dialog wieder abspielen (solange sich Oberfläche nicht zu stark verändert)
- Werkzeuge für Last- und **Performanztests** (dynamische Analysen): Laufzeitmessungen, Speicherplatzverbrauch, ...
- **Komparatoren**: vergleichen erwartete mit tatsächlichen Ergebnissen, filtern unwesentliche Details, können ggf. auch Zustand von Benutzeroberflächen prüfen
- Werkzeuge zur **Überdeckungsanalyse**: für gewählte Überdeckungsmetrik wird Buch darüber geführt, wieviel Prozent Überdeckung erreicht ist, welche Programmausschnitte noch nicht überdeckt sind
- [Werkzeuge für **statische Analysen**: Berechnung von Metriken, Kontroll- und Datenflussanomalien, ...]

4.8 Zusammenfassung

Dynamische Programmanalysen und das „traditionelle“ Testen sind die wichtigsten Mittel der analytischen Qualitätssicherung. Mindestens folgende Maßnahmen sollten immer durchgeführt werden:

- Suche nach Speicherlecks und Zugriff auf nichtinitialisierte Speicherbereiche (oder falsche Speicherbereiche) mit geeigneten Werkzeugen
- automatische Regressions-Testdurchführung mit entsprechenden Frameworks zur Testautomatisierung
- Überprüfung der Zweigüberdeckung durch Testfälle anhand von Kontrollfluss-graph (durch Werkzeuge)
- Verwendung von „Black-Box“-Testverfahren mit Äquivalenzklassenbildung für Eingabeparameter (Objekte) zur Bestimmung von Testfällen
- Einbau möglichst vieler abschaltbarer Konsistenzprüfungen in Code (Vor- und Nachbedingungen) und ggf. defensive Programmierung

4.9 Ergänzende Literatur

[Bi00] R. V. Binder: Testing Object-Oriented Systems, Addison-Wesley (2000), 1191 Seiten

[BP84] V.R. Basili, B.T. Perricone: *Software Errors and Complexity: An Empirical Investigation*, Communications of the ACM, Vol. 27, No. 1, 42-52, ACM Press (1984)

[FRSW03] F. Fraikin, E.H. Riedemann, A. Spillner, M. Winter: *Basiswissen Softwaretest - Certified Tester*, Skript zu Vorlesungen an der TU Darmstadt, Uni Dortmund, Bremen und FH Köln (2003)

[FS98] M. Fowler, K. Scott: *UML konzentriert: Die neue Standard-Objektmodellierungssprache anwenden*, Addison Wesley (1998), 188 Seiten

[MS01] J. D. McGregor, D. A. Sykes: *A Practical Guide to Testing Object-Oriented Software*, Addison-Wesley (2001)

[SL12] A. Spillner, T. Linz: ***Basiswissen Softwaretest***, dpunkt.verlag (2012; 5. Auflage), 290 Seiten

[Vi05] U. Vigenschow: *Objektorientiertes Testen und Testautomatisierung in der Praxis*, dpunkt.verlag (2005), 331 Seiten

5. Management der Software-Entwicklung

Themen dieses Kapitels:

- bessere/modernere Prozessmodelle
- Verbesserung/Qualität von Softwareprozessmodellen
- Projektmanagement(-werkzeuge)

Achtung:

Viele im folgenden vorgestellten Überlegungen sind nicht ausschließlich für **Software**-Entwicklungsprozesse geeignet, sondern werden ganz allgemein für die Steuerung komplexer technischer Entwicklungsprozesse eingesetzt.

Aufgaben des Managements:

- **Planungsaktivitäten**: Ziele definieren, Vorgehensweisen auswählen, Termine festlegen, Budgets vorbereiten, …
  - ⇨ Vorgehensmodelle, Kostenschätzung, Projektpläne
- **Organisationsaktivitäten**: Strukturieren von Aufgaben, Festlegung organisatorischer Strukturen, Definition von Qualifikationsprofilen für Positionen, …
  - ⇨ Rollenmodelle, Team-Modelle, Projektpläne
- **Personalaktivitäten**: Positionen besetzen, Mitarbeiter beurteilen, weiterbilden, …
  - ⇨ nicht Thema dieser Vorlesung
- **Leitungsaktivitäten**: Mitarbeiter führen, motivieren, koordinieren, …
  - ⇨ nicht Thema dieser Vorlesung
- **Kontrollaktivitäten**: Prozess- und Produktstandards entwickeln, Berichts- und Kontrollwesen etablieren, Prozesse und Produkte vermessen, Korrekturen, …
  - ⇨ Qualititätsmanagement, insbesondere für Software-Entwicklungsprozesse

Ziele des Managements:

Hauptziel des Projektmanagements ist die **Erhöhung der Produktivität**!

Allgemeine Definition von Produktivität:

Produktivität = Produktwert / Aufwand (oder: Leistung / Aufwand)

Für Software-Entwicklung oft verwendete Definition:

Produktivität = Größe der Software / geleistete Mitarbeitertage

Probleme mit dieser Definition:

- ⇨ Maß für Größe der Software
- ⇨ Berücksichtigung der Produktqualität
- ⇨ Aufwand = Mitarbeitertage ?
- ⇨ Nutzen (Return Of Investment) = Größe der Software ?

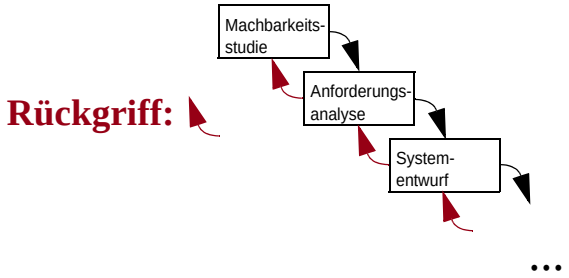
Einflussfaktoren für Produktivität [ACF97]:

Angabe der Form “+ 1 : X” steht für Produktivitätssteigerung um maximal Faktor X
Angabe der Form “- 1 : Y” steht für Produktivitätsminderung um maximal Faktor Y.

- ⇨ Werkzeug-Einsatz: + 1 : 1,6
- ⇨ Geeignete Methoden: + 1 : 1,9
- ⇨ Produktkomplexität: - 1 : 2,4
- ⇨ Hohe Zuverlässigkeitsanforderung: - 1 : 1,9
- ⇨ Firmenkultur (corporate identity): **+ 1 : 11**
- ⇨ Arbeitsumgebung (eigenes Büro, abschaltbares Telefon, …): + 1 : ???
- ⇨ Begabung der Mitarbeiter: **+ 1 : 10**
- ⇨ Erfahrung im Anwendungsgebiet: + 1 : 1,6
- ⇨ Bezahlung, Berufserfahrung: **kein messbarer Einfluss**

5.1 „Neuere“ Vorgehensmodelle

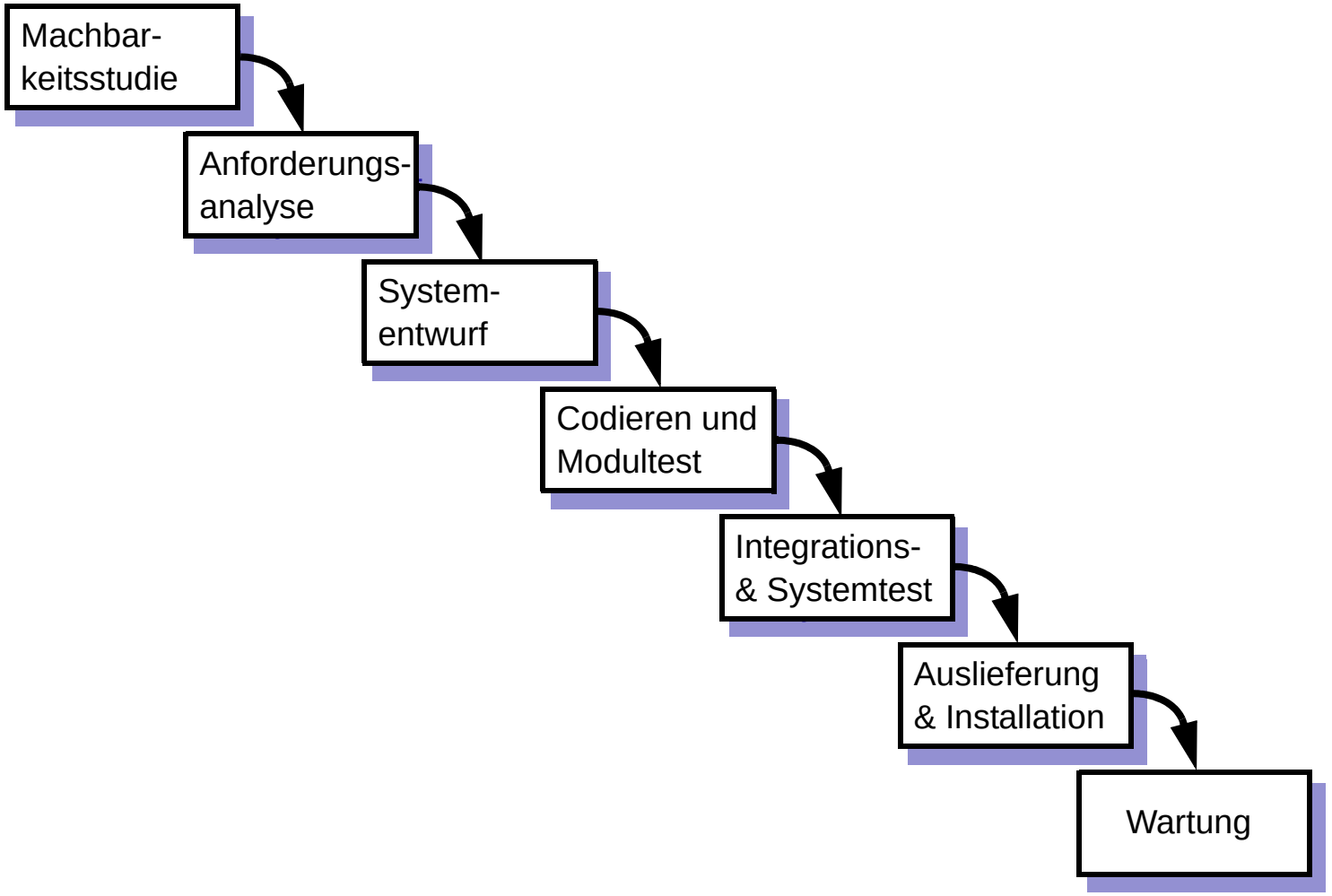
Die naheliegendste Idee zur Verbesserung des Wasserfallmodells ergibt sich durch die Einführung von **Zyklen** bzw. **Rückgriffen**. Sie erlauben Wiederaufnehmen früherer Phasen, wenn in späteren Phasen Probleme auftreten.



Weitere Vorgehensmodelle:

- das V-Modell (umgeklapptes Wasserfallmodell)
- das evolutionäre Modell (iteriertes Wasserfallmodell)
- Rapid Prototyping (Throw-Away-Prototyping)

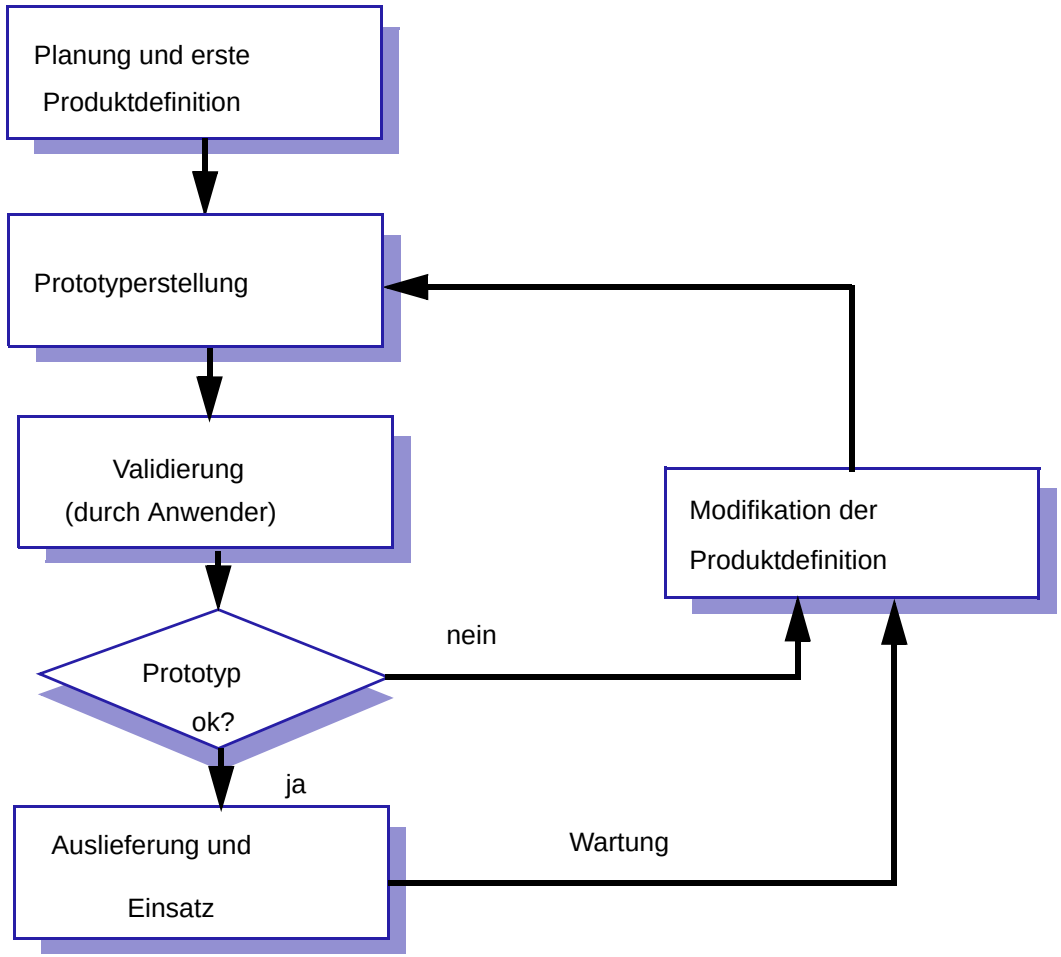
Zur Erinnerung - das Wasserfallmodell



Probleme mit dem Wasserfallmodell insgesamt:

- ☹ Wartung mit ca. 60% des Gesamtaufwandes ist eine Phase
  - ⇨ andere Prozessmodelle mit Wartung als eigener Entwicklungsprozess
- ☹ zu Projektbeginn sind nur ungenaue Kosten- und Ressourcenschätzungen möglich
  - ⇨ Methoden zur Kostenschätzung anhand von Lastenheft (Pflichtenheft)
- ☹ ein Pflichtenheft kann nie den Umgang mit dem fertigen System ersetzen, das erst sehr spät entsteht (Risikomaximierung)
  - ⇨ andere Prozessmodelle mit Erstellung von Prototypen, …
- ☹ Anforderungen werden früh eingefroren, notwendiger Wandel (aufgrund organisatorischer, politischer, technischer, … Änderungen) nicht eingeplant
  - ⇨ andere Prozessmodelle mit evolutionärer Software-Entwicklung
- ☹ strikte Phaseneinteilung ist unrealistisch (Rückgriffe sind notwendig)
  - ⇨ andere Prozessmodelle mit iterativer Vorgehensweise

Evolutionäres Modell (evolutionäres Prototyping):



Bewertung des evolutionären Modells:

- ☺ es ist sehr früh ein (durch Kunden) evaluierbarer Prototyp da
- ☺ Kosten und Leistungsumfang des gesamten Softwaresystems müssen nicht zu Beginn des Projekts vollständig festgelegt werden
- ☺ Projektplanung vereinfacht sich durch überschaubarere Teilprojekte
- ☺ Systemarchitektur muss auf Erweiterbarkeit angelegt sein
- ☹ es ist schwer, Systemarchitektur des ersten Prototypen so zu gestalten, dass sie alle später notwendigen Erweiterungen erlaubt
- ☹ Prozess der Prototyperstellung nicht festgelegt
  - ⇨ Spiralmodell von Berry Böhm integriert Phasen des Wasserfallmodells
- ☹ evolutionäre Entwicklung der Anforderungsdefinition birgt Gefahr in sich, dass bereits realisierte Funktionen hinfällig werden
- ☹ Endresultat sieht ggf. wie Software nach 10 Jahren Wartung aus

Rapid Prototyping (Throw-Away-Prototyping):

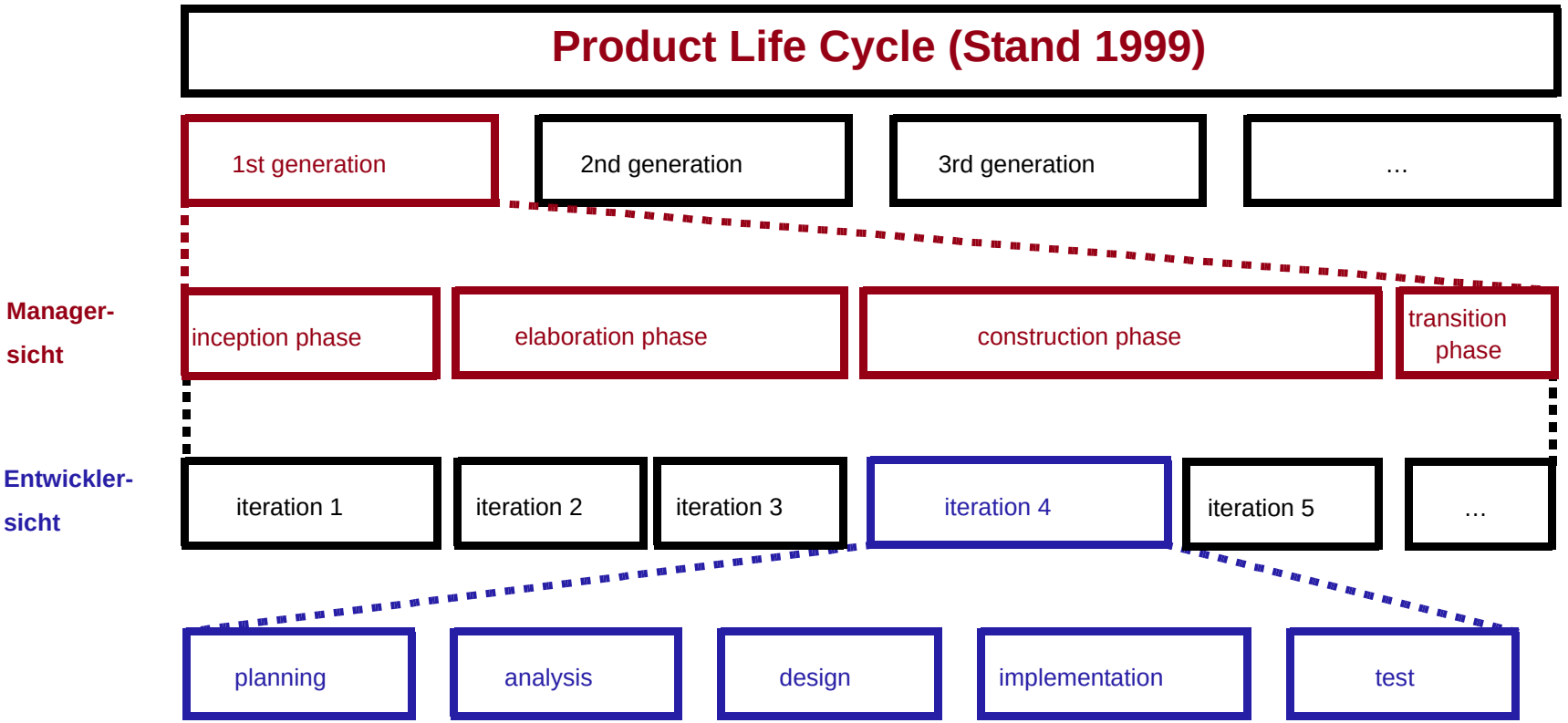
Mit Generatoren, ausführbaren Spezifikationssprachen, Skriptsprachen etc. wird

- ☞ Prototyp des Systems (seiner Benutzeroberfläche) realisiert
- ☞ dem Kunden demonstriert
- ☞ und anschließend weggeschmissen

Bewertung:

- ☺ erlaubt schnelle Klärung der Funktionalität und Risikominimierung
- ☺ Vermeidung von Missverständnissen zwischen Entwickler und Auftraggeber
- ☺ früher Test der Benutzerschnittstelle

5.2 Rational Unified Process für UML



☞ Firma IBM (ehemals Rational) dominiert(e) Entwicklung der Standard-OO-Modellierungssprache UML und des zugehörigen Vorgehensmodells.

Phasen der Lebenszyklusgenerationen:

- **Inception (Vorbereitung)**: Definition des Problembereichs und Projektziels für Produktgeneration mit Anwendungsbereichsanalyse (Domain Analysis) und Machbarkeitsstudie (für erste Generation aufwändiger)

  ⇨ bei Erfolg weiter zu …

- **Elaboration (Entwurf)**: erste Anforderungsdefinition für Produktgeneration mit grober Softwarearchitektur und Projektplan (ggf. mit Rapid Prototyping)

  ⇨ bei Erfolg weiter zu …

- **Construction (Konstruktion)**: Entwicklung der neuen Produktgeneration als eine Abfolge von Iterationen mit Detailanalyse, -design, … (wie beim evolutionären Vorgehensmodell)

  ⇨ bei Erfolg weiter zu …

- **Transition (Einführung)**: Auslieferung des Systems an den Anwender (inklusive Marketing, Support, Dokumentation, Schulung, …)

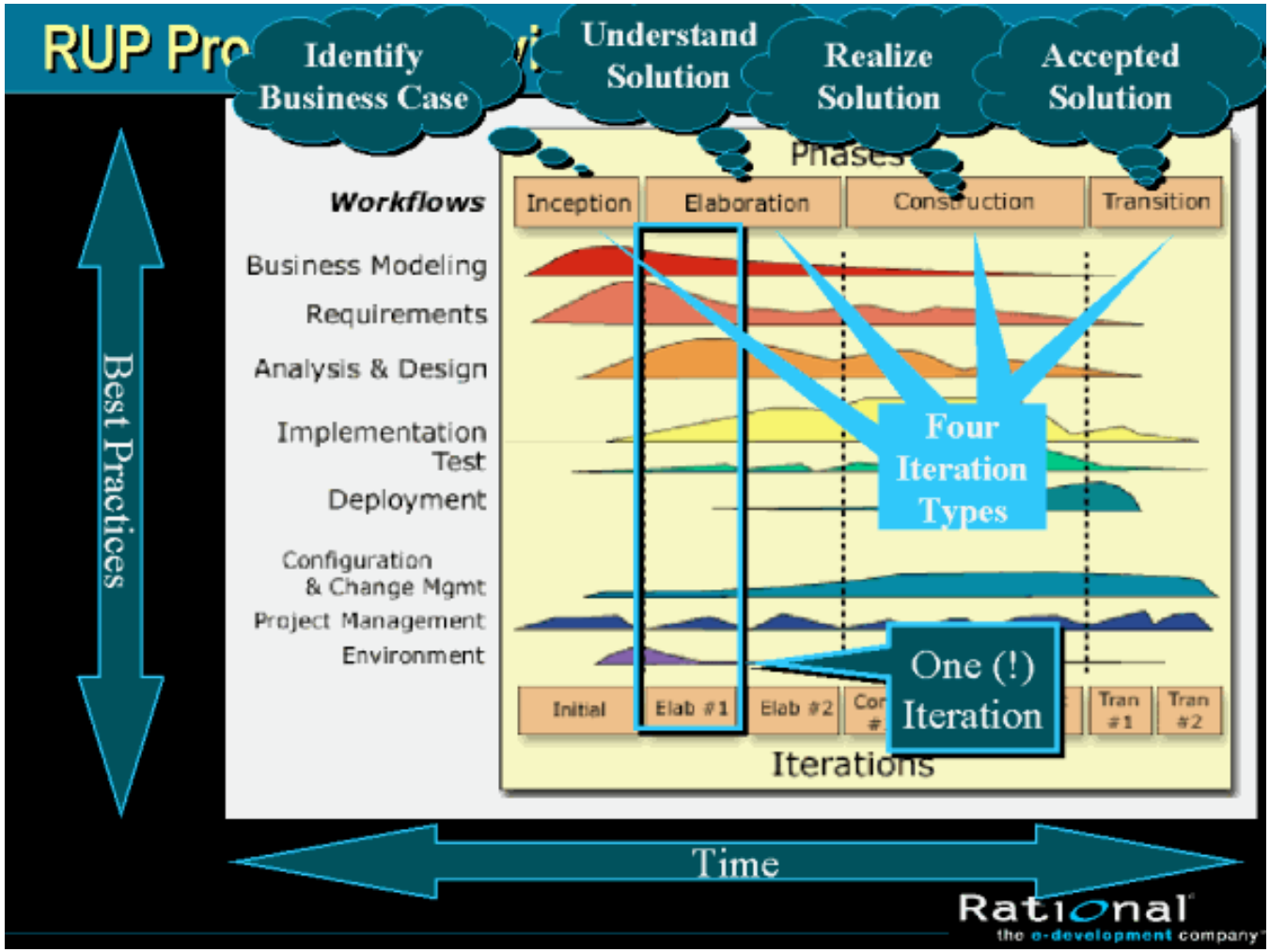
Eigenschaften des Rational Unified Prozesses (RUP):

- **modellbasiert**: für die einzelnen Schritte des Prozesses ist festgelegt, welche Modelle (Dokumente) des Produkts zu erzeugen sind
- **prozessorientiert**: die Arbeit ist in eine genau definierte Abfolge von Aktivitäten unterteilt, die von anderen Teams in anderen Projekten wiederholt werden können.
- **iterativ und inkrementell**: die Arbeit ist in eine Vielzahl von Iterationen unterteilt, das Produkt wird inkrementell entwickelt.
- **risikobewusst**: Aktivitäten mit hohem Risiko werden identifiziert und in frühen Iterationen in Angriff genommen.
- **zyklisch**: die Produktentwicklung erfolgt in Zyklen (Generationen). Jeder Zyklus liefert eine neue als kommerzielles Produkt ausgelieferte Systemgeneration.
- **ergebnisorientiert**: jede Phase (Iteration) ist mit der Ablieferung eines definierten Ergebnisses meist zu einem konkreten Zeitpunkt (Meilenstein) verbunden

Faustregeln für die Ausgestaltung eines Entwicklungsprozesses:

- die Entwicklung einer **Produktgeneration** dauert höchstens 18 Monate
- eine **Vorbereitungsphase** dauert 3-6 Wochen und besteht aus einer Iteration
- eine **Entwurfsphase** dauert 1-3 Monate und besteht aus bis zu 2 Iterationen
- eine **Konstruktionsphase** dauert 1-9 Monate und besteht aus bis zu 7 Iterationen
- eine **Einführungsphase** dauert 4-8 Wochen und besteht aus einer Iteration
- jede **Iteration** dauert 4-8 Wochen (ggf. exklusive Vorbereitungs- und Nachbereitungszeiten, die mit anderen Iterationen überlappen dürfen)
- das gewünschte **Ergebnis** (Software-Release) einer Iteration ist spätestens bei ihrem Beginn festgelegt (oft Abhängigkeit von Ergebnissen vorheriger Iterationen)
- die **geplante Zeit** für eine Iteration wird nie (höchstens um 50%) überschritten
- innerhalb der Konstruktionsphase wird mindestens im wöchentlichen Abstand ein **internes Software-Release** erstellt
- mindestens 40% **Reserve** an Projektlaufzeit für unbekannte Anforderungen, …

Arbeitsbereiche (Workflows) im RUP:



siehe RUP-Resource-Center: http://www.rational.net/rupcenter

Anmerkungen zu den Arbeitsbereichen (Workflows) des RUP:

- **Business Modeling** befasst sich damit, das Umfeld des zu erstellenden Software-systems zu erfassen (Geschäftsvorfälle, Abläufe, …)
- **Requirements (Capture)** befasst sich damit die Anforderungen an ein Software-system noch sehr informell zu erfassen
- **Analysis/Design** präzisiert mit grafischen Sprachen (Klassendiagramme etc.) die Anforderungen und liefert Systemarchitektur
- **Implementation/Test** entspricht den Aktivitäten in den Phasen „Codierung bis Integrationstest” des Wasserfallmodells
- **Deployment** entspricht „Auslieferung und Installation” des Wasserfallmodells
- **Configuration Management** befasst sich mit der Verwaltung von Softwareversionen und -varianten
- **Project Management** mit der Steuerung des Entwicklungsprozesses selbst
- **Environment** bezeichnet die Aktivitäten zur Bereitstellung benötigter Ressourcen (Rechner, Werkzeuge, …)

Bewertung des (Rational) Unified Prozesses:

- ☺ Manager hat die grobe “Inception-Elaboration-Construction-Transition”-Sicht
- ☺ Entwickler hat zusätzlich die feinere arbeitsbereichsorientierte Sicht
- ☺ Wartung ist eine Abfolge zu entwickelnder Produktgenerationen
- ☺ es wird endgültig die Illusion aufgegeben, dass Analyse, Design, … zeitlich begrenzte strikt aufeinander folgende Phasen sind
- ☹ komplexes noch in Veränderung befindliches Vorgehensmodell für UML
- ☹ noch nicht mit Behördenstandards (V-Modell, …) richtig integriert
- ☹ Qualitätssicherung ist kein eigener Aktivitätsbereich

5.3 Leichtgewichtige Prozessmodelle

Herkömmlichen Standards für Vorgehensmodelle zur Softwareentwicklung (V-Modell, RUP) wird vorgeworfen, dass

- sie sehr starr sind
- Unmengen an Papier produzieren
- und nutzlos Arbeitskräfte binden

Deshalb werden seit einiger Zeit sogenannte „leichtgewichtige“ Prozessmodelle (**light-weight processes**) propagiert, die sinnlosen bürokratischen Overhead vermeiden wollen:

- **Extreme Programming (XP)**: radikale Abkehr von bisherigen Vorgehens-modellen und Ersatz durch „best practices“ der Programmierung [Be99]
- **Agile Prozessmodelle**: Versuche, herkömmliche Prozessmodelle auf das unbedingt notwendige zurückzuschneiden und situationsbedingt flexibel und schnell (agil) voranzuschreiten

Grundideen von XP:

- Verzicht auf Phasen der Softwareentwicklung, Arbeitsbereiche, …
- keine eigenständigen Analyse- oder Designaktivitäten vor der Codierung
- keine Erstellung getrennter Dokumentationsdokumente, Modelle, …
- der (gut kommentierte) Code ist seine eigene Dokumentation
- aber großes Gewicht auf sorgfältige Testplanung
- Tests werden immer zusammen mit (oder gar vor) Code geschrieben
- nach jeder Änderung werden alle Tests durchlaufen (Regressionstests)
- unbedingtes Einhalten der 40-Stunden-Woche
- Auftraggeber soll permanent neue Anforderungen aufstellen und priorisieren
- sehr kurze Iterationszyklen, häufige Releases
- „Pair Programming“ Teams
- Subsets obiger Ideen dürfen nicht XP genannt werden

Randbedingungen:

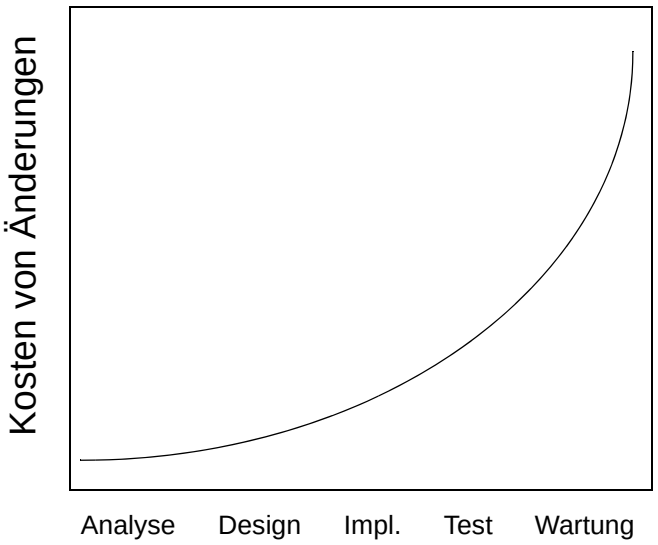
- maximal 5 bis 10 Programmierer an einem Projekt beteiligt
  - ⇨ Beschränkung auf relativ kleine Projekte
- Kommunikation zwischen Programmierern und mit dem Kunden sehr intensiv
  - ⇨ keine räumlich verteilten Teams
- Kunde verzichtet auf separate Dokumentation der Software
  - ⇨ zugunsten einer durch ausführliches Testen unterstützten Rapid-Prototyping-Vorgehensweise
- Programmierer versuchen nicht, bei der Realisierung des aktuellen Releases bereits künftige Releases mit zu berücksichtigen
- Programmierer sind aber zum ständigen Umbau (Redesign) der bereits erstellten Software bereit
  - ⇨ Prinzip des Refactorings von Software

Bewertung von XP:

- ☺ systematisches Hacking: Kompromiss zwischen Wirklichkeit der Programmentwicklung in vielen Fällen und sehr aufwändigen Vorgehensmodellen
- ☺ Betonung der menschlichen Komponente: 40-Stunden-Woche, intensive Kommunikation, Pair Programming, Empfehlungen für Arbeitsumgebung
- ☺ systematisches evolutionäres Rapid Prototyping: durch Testen und Refactoring (Software-Sanierung in kleinen Schritten) versucht man Nachteile zu vermeiden
- ☹ Endprodukt besitzt (ausser Kommentaren im Quellcode und Testfälle) keine Dokumentation
- ☹ Entwicklung des benötigten Gesamtsystems wird durch schnelle Produktion vieler kleiner Releases nicht unbedingt sehr zielgerichtet sein
- ☹ Grundannahmen (z.B. leichte Änderbarkeit des so erstellten Codes über gesamte Projektlaufzeit hinweg) nicht hinreichend empirisch belegt
- ☹ nur für kleine Projekte in nicht sicherheitskritschen Anwendungsbereichen

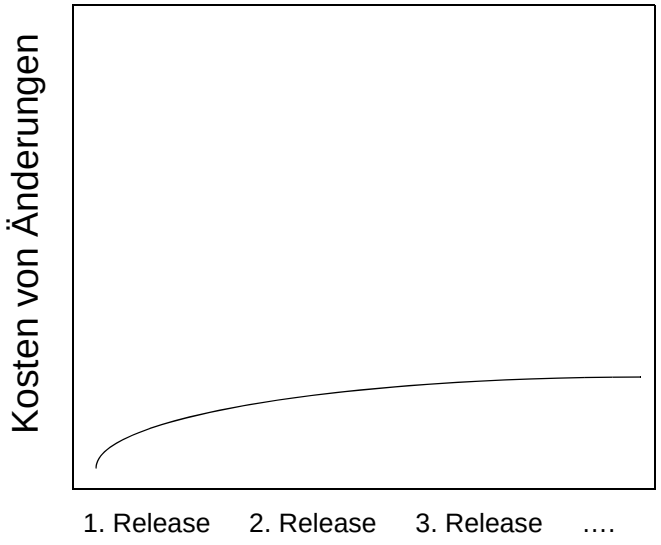
Lineare Kostenkurve:

Wasserfallmodell



zeitlicher Projektverlauf
(tatsächlich)

XP-Ansatz



zeitlicher Projektverlauf
(vermutet?)

Bei „klassischen“ Projekten steigen die Kosten für geänderte Anforderungen mit Abschluss jeder Phase deutlich an, bei XP verspricht man sich eine konstante Kostenkurve für die Durchführung gewünschter Änderungen (da nur Code zu ändern ist)

5.4 Verbesserung der Prozessqualität

Ausgangspunkt der hier vorgestellten Ansätze sind folgende Überlegungen:

- ⇨ Softwareentwicklungsprozesse sind selbst Produkte, deren Qualität überwacht und vTQMerbessert werden muss
- ⇨ bei der Softwareentwicklung sind bestimmte Standards einzuhalten (Entwicklungsprozess muss dokumentiert und nachvollziehbar sein)
- ⇨ es bedarf kontinuierlicher Anstregungen, um die Schwächen von Entwicklungsprozessen zu identifizieren und zu eliminieren

Hier vorgestellte Ansätze:

- ⇨ **ISO 9000 Normenwerk** (Int. Standard für die Softwareindustrie)
- ⇨ **Capability Maturity Model** (CMM/CMMI) des Software Engineering Institutes (SEI) an der Carnegie Mellon University
- ⇨ ISO-Norm **SPiCE** (http://www.sqi.gu.edu.au/SPICE/) integriert und vereinheitlicht CMM und ISO 9000

Qualitätssicherung mit der ISO 9000:

Das **ISO 9000 Normenwerk** (http://www.iso-9000.co.uk/) legt für das Auftraggeber-Lieferantenverhältnis einen allgemeinen organisatorischen Rahmen zur Qualitätssicherung fest.

Das **ISO 9000 Zertifikat** bestätigt, dass die Verfahren eines Unternehmens der ISO 9000 Norm entsprechen.

Wichtige Teile:

- **ISO 9000-1**: allgemeine Einführung und Überblick
- **ISO 9000-3**: Anwendung von ISO 9001 auf Softwareproduktion
- **ISO 9001**: Modelle der Qualitätssicherung in Design/Entwicklung, Produktion, Montage und Kundendienst
- **ISO 9004**: Aufbau und Verbesserung eines Qualitätsmanagementsystems

Von ISO 9000-3 vorgeschriebene Dokumente:

- **Vertrag Auftraggeber - Lieferant**: Tätigkeiten des Auftraggebers, Behandlung von Anforderungsänderungen, Annahmekriterien (Abnahmetest), …
- **Spezifikation**: funktionale Anforderungen, Ausfallsicherheit, Schnittstellen, … des Softwareprodukts
- **Entwicklungsplan**: Zielfestlegung, Projektmittel, Entwicklungsphasen, Management, eingesetzte Methoden und Werkzeuge, …
- **Qualitätssicherungsplan**: Qualitätsziele (messbare Größen), Kriterien für Ergebnisse v. Entwicklungsphasen, Planung von Tests, Verifikation, Inspektionen
- **Testplan**: Teststrategie (für Integrationstest), Testfälle, Testwerkzeuge, Kriterien für Testvollständigkeit/Testende
- **Wartungsplan**: Umfang, Art der Unterstützung, …
- **Konfigurationsmanagement**: Plan für Verwaltung von Softwareversionen und Softwarevarianten

Von ISO 9000-3 vorgeschriebene Tätigkeiten:

- **Konfigurationsmanagement** für Identifikation und Rückverfolgung von Änderungen, Verwaltung parallel existierender Varianten
- **Dokumentmanagement** für geordnete Ablage und Verwaltung aller bei der Softwareentwicklung erzeugten Dokumente
- **Qualitätsaufzeichnungen** (Fehleranzahl oder Metriken) für Verbesserungen am Produkt und Prozess
- **Festlegung** von Regeln, Praktiken und Übereinkommen für ein Qualitäts-sicherungssystem
- **Schulung** aller Mitarbeiter sowie Verfahren zur Ermittlung des Schulungsbedarfes

Zertifizierung:

Die Einhaltung der Richtlinien der Norm wird von unabhängigen Zertifizierungs-stellen im jährlichen Rhythmus überprüft.

Bewertung von ISO 9000:

- ☺ lenkt die Aufmerksamkeit des Managements auf Qualitätssicherung
- ☺ ist ein gutes Marketing-Instrument
- ☺ reduziert das Produkthaftungsrisiko (Nachvollziehbarkeit von Entscheidungen)
- ☹ Nachvollziehbarkeit und Dokumentation von Prozessen reicht aus
- ☹ keine Aussage über Qualität von Prozessen und Produkten
- ☹ (für kleine Firmen) nicht bezahlbarer bürokratischer Aufwand
- ☹ Qualifikation der Zertifizierungsstellen umstritten
- ☹ oft große Abweichungen zwischen zertifiziertem Prozess und realem Prozess

Das Capability Maturity Model (CMM):

Referenzmodell zur Beurteilung von Softwarelieferanten, vom Software Engineering Institute entwickelt (http://www.sei.cmu.edu/cmm/cmms.html).

- ⇨ Softwareentwicklungsprozesse werden in **5 Reifegrade** unterteilt
- ⇨ Reifegrad (**maturity**) entspricht Qualitätsstufe der Softwareentwicklung
- ⇨ höhere Stufe beinhaltet Anforderungen der tieferen Stufen

Stufe 1 - chaotischer initialer Prozess (ihr Stand vor dieser Vorlesung):

- ❑ Prozess-Charakteristika:
  - ⇨ unvorhersehbare Entwicklungskosten, -zeit und -qualität
  - ⇨ kein Projektmanagement, nur „Künstler“ am Werke
- ❑ notwendige Aktionen:
  - ⇨ Planung mit Kosten- und Zeitschätzung einführen
  - ⇨ Änderungs- und Qualitätssicherungsmanagement

Stufe 2 - wiederholbarer intuitiver Prozess (Stand nach dieser Vorlesung?):

- Prozess-Charakteristika:
  - ⇨ Kosten und Qualität schwanken, gute Terminkontrolle
  - ⇨ Know-How einzelner Personen entscheidend
- notwendige Aktionen:
  - ⇨ Prozessstandards entwickeln
  - ⇨ Methoden (für Analyse, Entwurf, Testen, …) einführen

Stufe 3 - definierter qualitativer Prozess (Stand der US-Industrie 1989?):

- Prozess-Charakteristika:
  - ⇨ zuverlässige Kosten- und Terminkontrolle, schwankende Qualität
  - ⇨ institutionalisierter Prozess, unabhängig von Individuen
- notwendige Aktionen:
  - ⇨ Prozesse vermessen und analysieren
  - ⇨ quantitative Qualitätssicherung

Stufe 4 - gesteuerter/geleiteter quantitativer Prozess:

- Prozess-Charakteristika:
  - ⇨ gute statistische Kontrolle über Produktqualität
  - ⇨ Prozesse durch Metriken gesteuert
- notwendige Aktionen:
  - ⇨ instrumentierte Prozessumgebung (mit Überwachung)
  - ⇨ ökonomisch gerechtfertigte Investitionen in neue Technologien

Stufe 5 - optimierender rückgekoppelter Prozess:

- Prozess-Charakteristika:
  - ⇨ quantitative Basis für Kapitalinvestitionen in Prozessautomatisierung und -verbesserung
- notwendige Aktionen:
  - ⇨ kontinuierlicher Schwerpunkt auf Prozessvermessung und -verbesserung (zur Fehlervermeidung)

Stand von Organisationen im Jahre 2000 (2007):

Daten vom Software Engineering Institute (SEI) aus dem Jahr 2000 (2007) unter
http://www.sei.cmu.edu/appraisal-program/profile/profile.html
(Die Daten in Klammern betreffen das Jahr 2007 für den Vergleich mit dem Stand im Jahr 2000; die genannte URL existiert nicht mehr):

- 32,3 % (1,7 %) der Organisationen im Zustand „initial“
- 39,3 % (32,7 %) der Organisationen im Zustand „wiederholbar“
- 19,4 % (36,1 %) der Organisationen im Zustand „definiert“
- 5,4 % (4,2 %) der Organisationen im Zustand „kontrolliert“
- 3,7 % (16,4 %) der Organisationen im Zustand „optimierend“

Genauere Einführung in CMM findet man in [Dy02]; weitere Informationen zum Nachfolger CMMI von CMM (siehe folgende Seiten) findet man unter:

http://cmmiinstitute.com/

ISO 9000, TQM und CMM im Vergleich:

- Schwerpunkt der **ISO 9001** Zertifizierung liegt auf Nachweis eines Qualitäts-managementsystems im Sinne der Norm
  - ⇨ allgemein für Produktionsabläufe geeignet
  - ⇨ genau ein Reifegrad wird zertifiziert
- **CMM** konzentriert sich auf Qualitäts- und Produktivitätssteigerung des gesamten Softwareentwicklungsprozesses
  - ⇨ auf Softwareentwicklung zugeschnitten
  - ⇨ dynamisches Modell mit kontinuierlichem Verbesserungsdruck
- **TQM** ist eine ganzheitliche, umfassende Unternehmensphilosophie, die Qualität aus der Sicht des Kunden als oberstes Ziel verfolgt
  - ⇨ umfassender Ansatz mit Berücksichtigung sozialer Aspekte
  - ⇨ kann ISO 9000 und CMM als Bausteine enthalten
- ISO-Norm **SPiCE** (http://www.sqi.gu.edu.au/SPICE/) integriert und vereinheitlicht CMM und ISO 9000 (als ISO/IEC 15504)

SPiCE = Software Process Improvement and Capability dEtermination:

Internationale Norm für **Prozessbewertung** (und Verbesserung). Sie bildet einheitlichen Rahmen für Bewertung der Leistungsfähigkeit von Organisationen, deren Aufgabe Entwicklung oder Erwerb, Lieferung, Einführung und Betreuung von Software-Systemen ist. Norm legt Evaluierungsprozess und Darstellung der Evaluierungsergebnisse fest.

Unterschiede zu CMM:

- orthogonale Betrachtung von Reifegraden und Aktivitätsbereichen
- deshalb andere Definition der 5 Reifegrade (z.B. „1“ = alle Aktivitäten eines Bereiches sind vorhanden, Qualität der Aktivitäten noch unerheblich, ...)
- jedem Aktivitätsbereich oder Unterbereich kann ein anderer Reifegrad zugeordnet werden

Aktivitätsbereiche von SPiCE:

- **Customer-Supplier-Bereich**:
  - ⇨ Aquisition eines Projektes (Angebotserstellung, ...)
  - ⇨ ...
- **Engineering-Bereich**:
  - ⇨ Software-Entwicklung (Anforderungsanalyse, ... , Systemintegration)
  - ⇨ Software-Wartung
- **Support-Bereich**:
  - ⇨ Qualitätssicherung
  - ⇨ ...
- **Management-Bereich**:
  - ⇨ Projekt-Management
  - ⇨ ...
- **Organisations-Bereich**:
  - ⇨ Prozess-Verbesserung
  - ⇨ ...

CMMI = Capability Maturity Model Integration (neue Version von CMM):

CMMI ist die **neue Version des Software Capability Maturity Model**. Es ersetzt nicht nur verschiedene Qualitäts-Modelle für unterschiedliche Entwicklungs-Disziplinen (z.B. für Software-Entwicklung oder System-Entwicklung), sondern integriert diese in einem neuen, modularen Modell. Dieses modulare Konzept ermöglicht zum einen die Integration weiterer Entwicklungs-Disziplinen (z.B. Hardware-Entwicklung), und zum anderen auch die Anwendung des Qualitätsmodells in übergreifenden Disziplinen (z.B. Entwicklung von Chips mit Software).

Geschichte von CMM und CMMI:

- ⇨ 1991 wird Capability Maturity Model 1.0 herausgegeben
- ⇨ 1993 wird CMM überarbeitet und in der Version 1.1 bereitgestellt
- ⇨ 1997 wird CMM 2.0 kurz vor Verabschiedung vom DoD zurückgezogen
- ⇨ 2000 wird CMMI als Pilotversion 1.0 herausgegeben
- ⇨ 2002 wird CMMI freigegeben
- ⇨ Ende 2003 ist die Unterstützung CMM ausgelaufen

Eigenschaften von CMMI:

Es gibt **Fähigkeitsgrade** für einzelne Prozessgebiete (ähnlich zu SPiCE):

0 - Incomplete:
Ausgangszustand, keine Anforderungen

1 - Performed:
die spezifischen Ziele des Prozessgebiets werden erreicht

2 - Managed:
der Prozess wird gemanagt

3 - Defined:
der Prozess wird auf Basis eines angepassten Standard-Prozesses gemanagt und verbessert

4 - Quantitatively Managed:
der Prozess steht unter statistischer Prozesskontrolle

5 - Optimizing:
der Prozess wird mit Daten aus der statistischen Prozesskontrolle verbessert

Eigenschaften von CMMI - Fortsetzung:

Es gibt **Reifegrade**, die Fähigkeitsgrade auf bestimmten Prozessgebieten erfordern (ähnlich zu CMM):

1- Initial:
keine Anforderungen, diesen Reifegrad hat jede Organisation automatisch

2 - Managed:
die Projekte werden gemanagt durchgeführt und ein ähnliches Projekt kann erfolgreich wiederholt werden

3 - Defined:
die Projekte werden nach einem angepassten Standard-Prozess durchgeführt, und es gibt eine kontinuierliche Prozessverbesserung

4 - Quantitatively Managed:
es wird eine statistische Prozesskontrolle durchgeführt

5 - Optimizing:
die Prozesse werden mit Daten aus statistischen Prozesskontrolle verbessert

Konsequenzen für die „eigene“ Software-Entwicklung:

Im Rahmen von Studienarbeiten, Diplomarbeiten, … können Sie keinen CMM(I)-Level-5- oder SPiCE-Software-Entwicklungsprozess verwenden, aber:

- Einsatz von Werkzeugen für **Anforderungsanalyse, Modellierung** und **Projektplanung**
  - ⇨ in der Vorlesung „Software-Engineering - Einführung“ behandelt
- Einsatz von **Konfigurations- und Versionsmanagement**-Software
  - ⇨ wird in dieser Vorlesung behandelt
- Einsatz von Werkzeugen für systematisches **Testen, Messen** der Produktqualität
  - ⇨ wird in dieser Vorlesung behandelt
- Einsatz von „**Extreme Programming**“-Techniken
  - ⇨ siehe „Software-Praktikum“ und z.B. [Be99] vom Erfinder Kent Beck
- Einsatz von Techniken zur Verbesserung des „persönlichen“ Vorgehensmodells
  - ⇨ siehe [Hu96] über den „**Personal Software Process**“
    (Buchautor Humphrey ist einer der „Erfinder“ von CMM)

5.5 Projektpläne und Projektorganisation

Am Ende der Machbarkeitsstudie steht die Erstellung eines Projektplans mit

- ⇨ Identifikation der einzelnen **Arbeitspakete**
- ⇨ **Terminplanung** (zeitliche Aufeinanderfolge der Pakete)
- ⇨ **Ressourcenplanung** (Zuordnung von Personen zu Paketen, …)

Hier wird am deutlichsten, dass eine Machbarkeitsstudie ohne ein grobes Design der zu erstellenden Software nicht durchführbar ist, da:

- ⇨ Arbeitspakete ergeben sich aus der Struktur der Software
- ⇨ Abhängigkeiten und Umfang der Pakete ebenso
- ⇨ Realisierungsart der Pakete bestimmt benötigte Ressourcen

Konsequenz: Projektplanung und -organisation ist ein fortlaufender Prozess.
Zu Projektbeginn hat man nur einen groben Plan, der sukzessive verfeinert wird.

Terminologie:

- **Prozessarchitektur** = grundsätzliche Vorgehensweise einer Firma für die Beschreibung von Software-Entwicklungsprozessen (Notation, Werkzeuge)
- **Prozessmodell** = Vorgehensmodell = von einer Firma gewählter Entwicklungs-prozess (Wasserfallmodell oder RUP oder …)
- **Projektplan** = an einem Prozessmodell sich orientierender Plan für die Durchfüh-rung eines konkreten Projektes
- **Vorgang** = Aufgabe = Arbeitspaket = abgeschlossene Aktivität in Projektplan, die
  - ⇨ bestimmte Eingaben (Vorbedingungen) benötigt und Ausgaben produziert
  - ⇨ Personal und (sonstige) Betriebsmittel für Ausführung braucht
  - ⇨ eine bestimmte Zeitdauer in Anspruch nimmt
  - ⇨ und Kosten verursacht und/oder Einnahmen bringt
- **Phase** = Zusammenfassung mehrerer zusammengehöriger Vorgänge zu einem globalen Arbeitsschritt
- **Meilenstein** = Ende einer Gruppe von Vorgängen (Phase) mit besonderer Bedeu-tung (für die Projektüberwachung) und wohldefinierten *Ergebnissen*

Beispielprojekt (Kundenbeschreibung der Anforderungen):

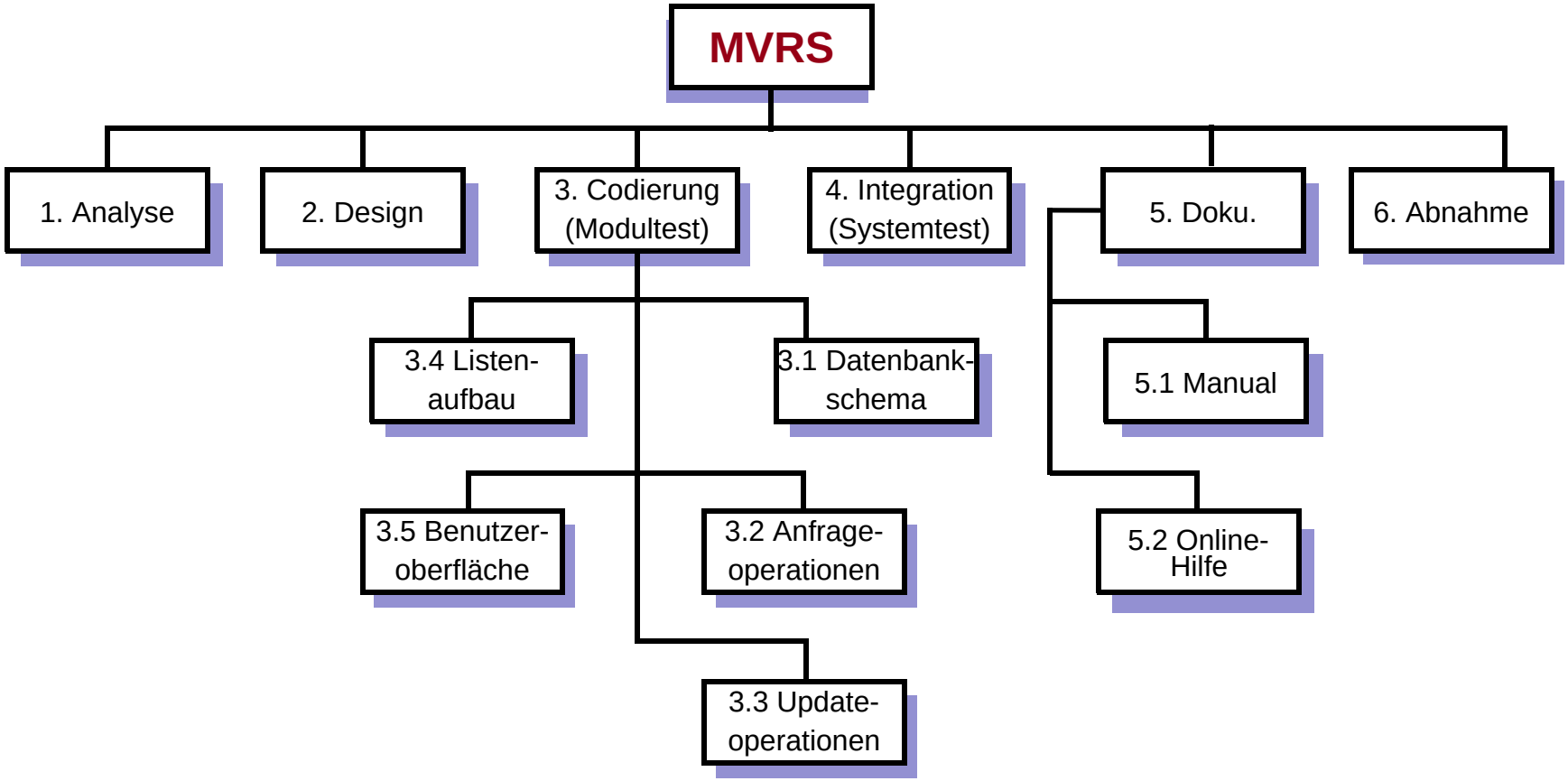
Motor Vehicle Reservation System (MVRS)

A rental office lends motor vehicles of different types. The assortment comprises cars, vans, and trucks. Vans are small trucks, which may be used with the same driving license as cars. Some client may reserve motor vehicles of a certain category for a certain period. He or she has to sign a reservation contract. The rental office guarantees that a motor vehicle of the desired category will be available for the requested period. The client may cancel the reservation at any time. When the client fetches the motor vehicle he or she has to sign a rental contract and optionally an associated insurance contract. Within the reserved period, at latest at its end, the client returns the motor vehicle and pays the bill.

Annahmen für MVRS-Projektplanung über (Grob-)Design der Software:

- es wird eine (sehr einfache) Dreischichtenarchitektur verwendet
- alle Daten werden in einer Datenbank gespeichert
  - ⇨ Datenbankschema muss entworfen werden
  - ⇨ Anfrageoperationen setzen auf Schema auf und geben Ergebnisse in Listenform aus
  - ⇨ Update-Operationen setzen auf Datenbankschema auf und sind unabhängig von Anfrageoperationen
- die Realisierung der Benutzeroberfläche ist von der Datenbank entkoppelt, für den sinnvollen Modultest der Operationen braucht man aber die Oberfläche
- um das Beispiel nicht zu kompliziert zu gestalten, wird das Wasserfallmodell mit Integration der einzelnen Teilsysteme im „Big Bang“-Testverfahren verwendet
- gedruckte Manuale und Online-Hilfe enthalten Screendumps der Benutzer-oberfläche (teilweise parallele Bearbeitung trotzdem möglich)

Aufteilung des MVRS-Projekts in Arbeitspakete (Aufgaben):



Im obigen Bild fehlen noch die Angaben für die geschätzten Arbeitszeiten zur Bearbeitung der einzelnen Teilaufgaben des „Motor Vehicle Reservation System“-Projekts

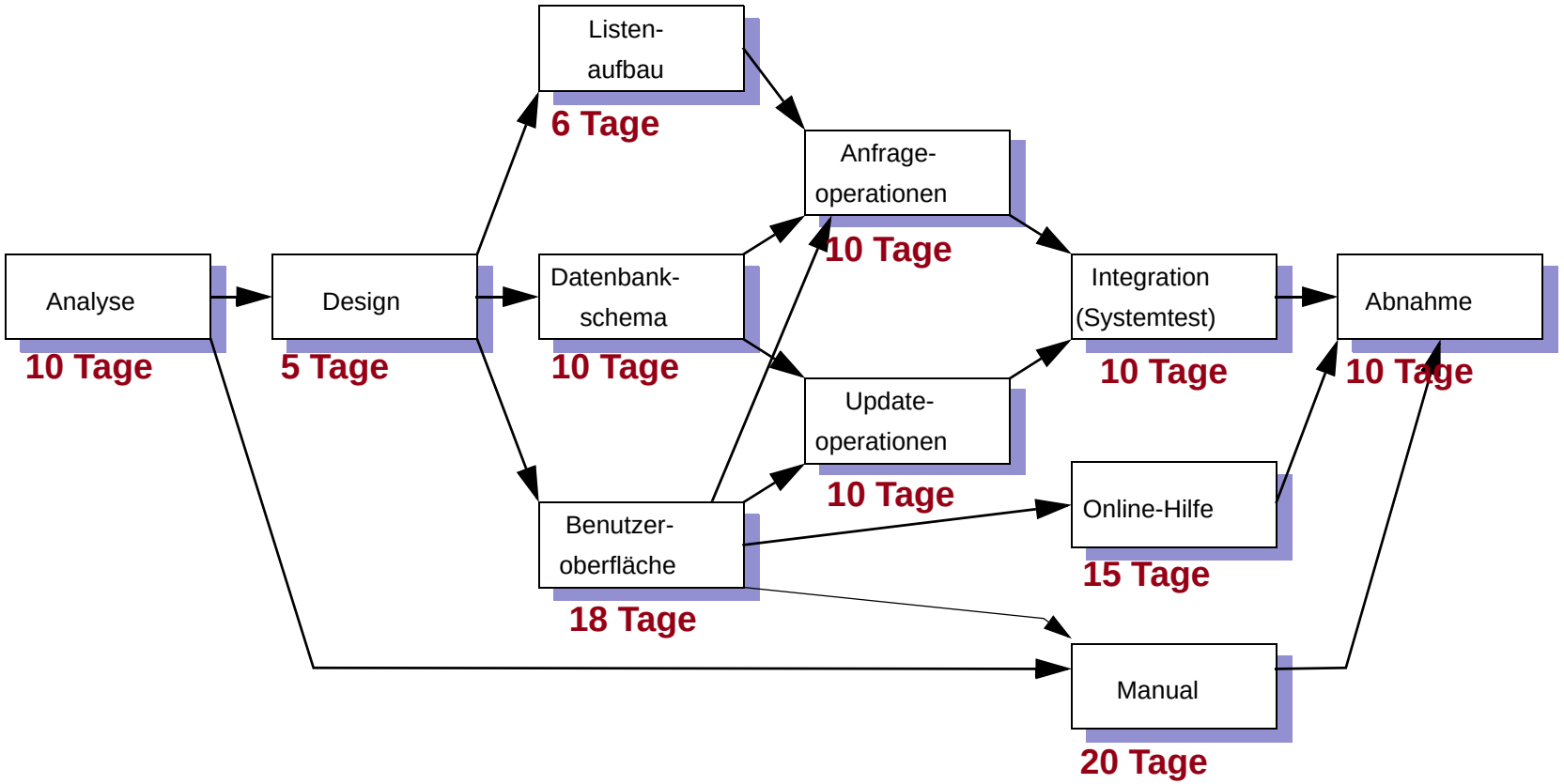
Organisation von Aufgaben mit MS Project:

ID	WBS	Task Name	Duration	Start	Finish	Predecessors
1	1	Analyse	10 days	Mon 14.04.03	Fri 25.04.03	
2	2	Design	5 days	Mon 28.04.03	Fri 02.05.03	1
3	3	Codierung	30 days	Mon 05.05.03	Fri 13.06.03	2
4	3.1	Code-Datenbankschema	10 days	Mon 05.05.03	Fri 16.05.03	
5	3.2	Code-Listenaufbau	6 days	Mon 19.05.03	Mon 26.05.03	
6	3.3	Code-Benutzeroberfläche	18 days	Mon 05.05.03	Wed 28.05.03	
7	3.4	Code-Update-Operationen	10 days	Mon 02.06.03	Fri 13.06.03	6;4
8	3.5	Code-Anfrageoperationen	10 days	Mon 02.06.03	Fri 13.06.03	4;5
9	4	Dokumentation	37 days	Thu 01.05.03	Fri 20.06.03	1
10	4.1	Doku-Manual	20 days	Thu 01.05.03	Wed 28.05.03	6I
11	4.2	Doku-Online-Hilfe	15 days	Mon 02.06.03	Fri 20.06.03	6
12	5	Integration	10 days	Mon 16.06.03	Fri 27.06.03	8;7;3
13	6	Abnahme	10 days	Mon 30.06.03	Fri 11.07.03	10;11;12

Aufgabenhierarchie Dauer Start/Ende Datum Abhängigkeiten

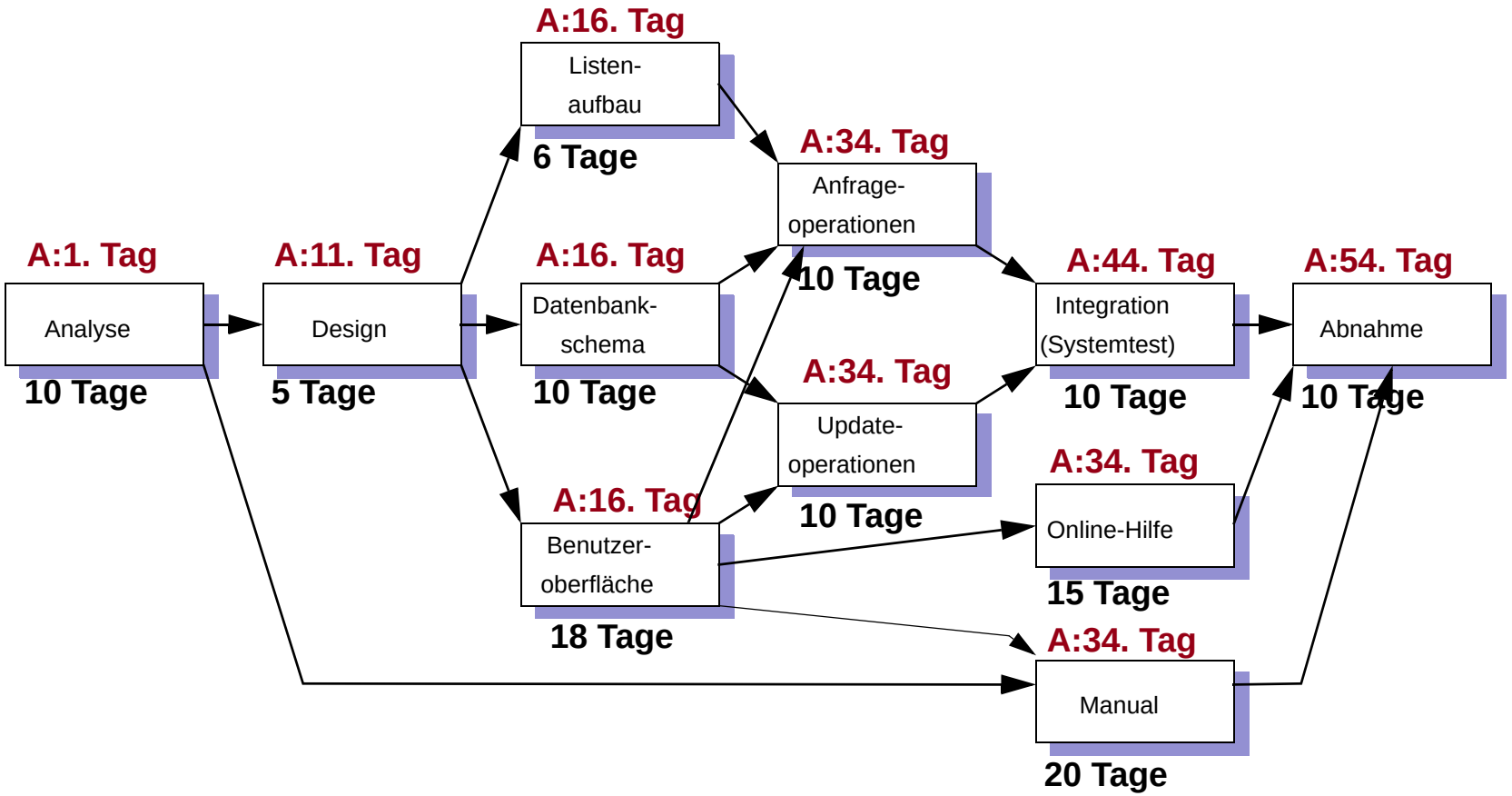
Achtung: Start und Ende von Aufgaben werden aus Dauer der Aufgaben und Zeitpunkt für Projektbeginn automatisch errechnet (siehe folgende Folien).

Planung von Arbeitspaketabhängigkeiten (mit PERT-Charts) - Idee:



PERT-Chart = **P**roject **E**valuation and **R**eview **T**echnique

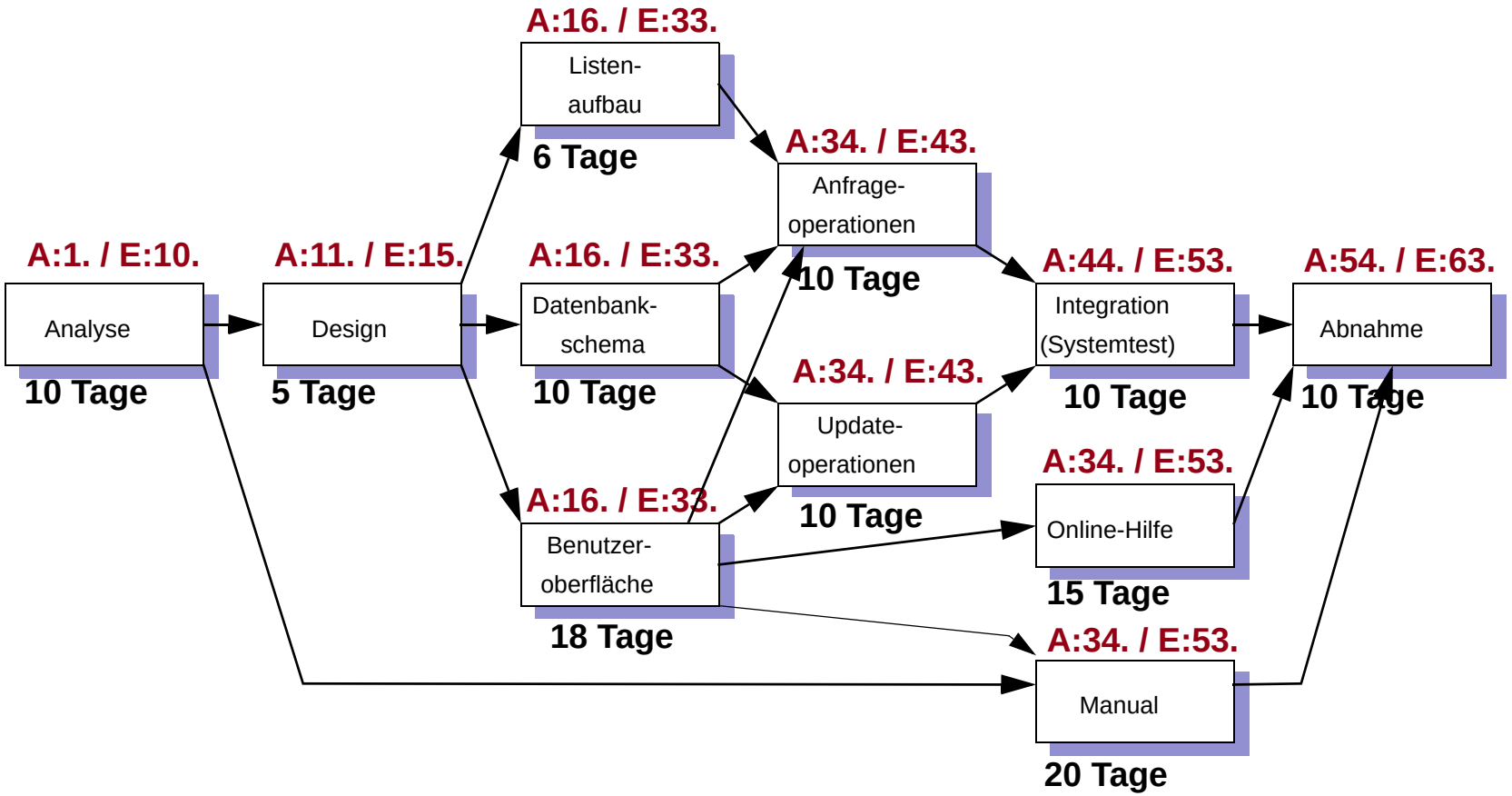
Planung von Arbeitspaketabhängigkeiten - früheste Anfangszeiten:



Vorwärtsberechnung von Startdatum für Aufgabe:

früheste Anfangszeit = Max(früheste Vorgängeranfangszeit + Vorgängerdauer)

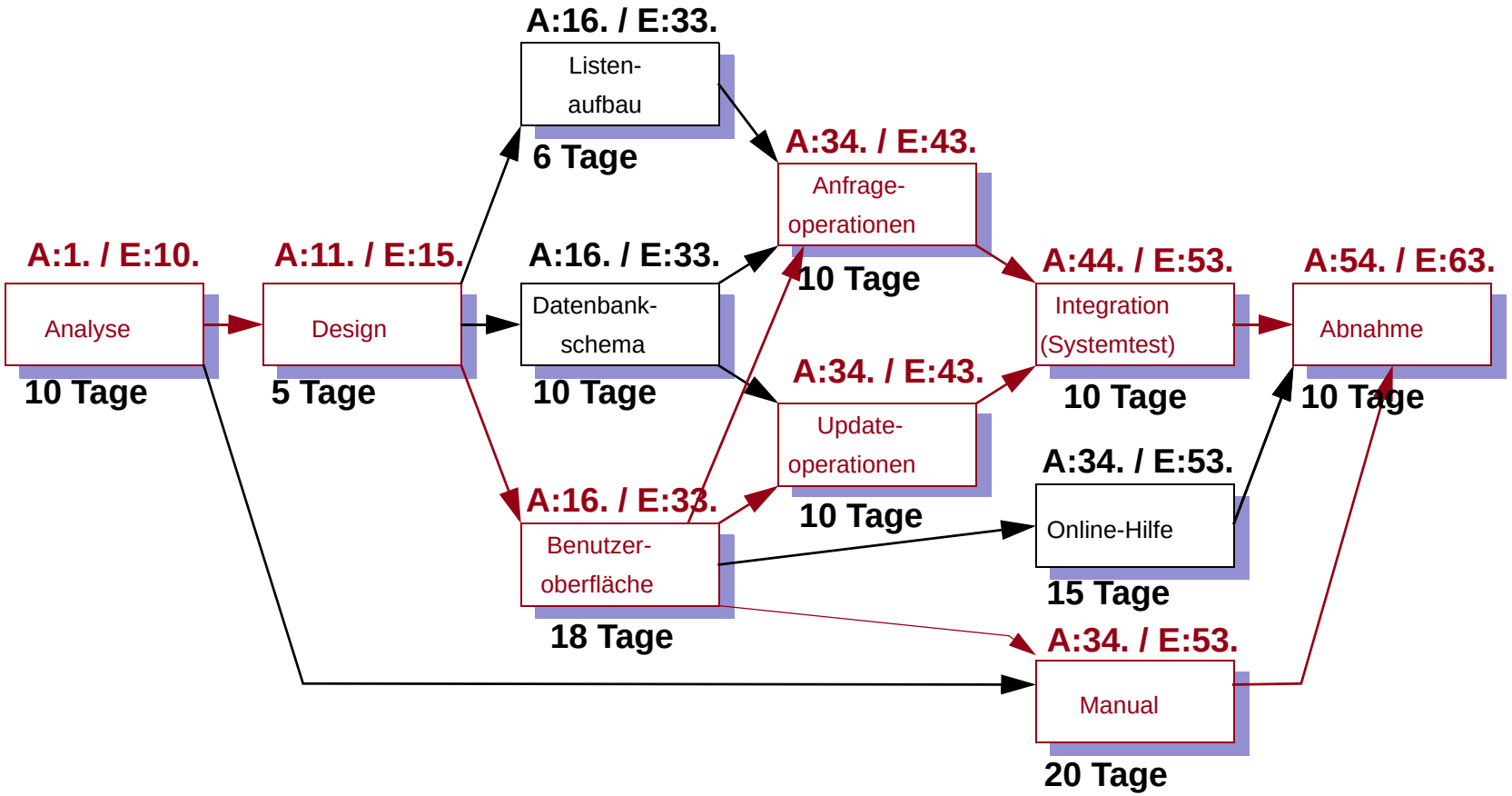
Planung von Arbeitspaketabhängigkeiten - späteste Endzeiten:



Rückwärtsberechnung von Enddatum für Aufgabe:

späteste Endzeit = Min(späteste Nachfolgerendzeit - Nachfolgerdauer)

Planung von Arbeitspaketabhängigkeiten - kritische Pfade u. Aufgaben:



Berechnung kritischer Pfade mit kritischen Aufgaben:

für **kritische Aufgabe** gilt: früheste Anfangszeit + Dauer = späteste Endzeit + 1

Zusammenfassung der Berechnung:

- **Vorwärtsberechnung** frühester Anfangszeiten für Aufgaben:
  - ⇨ früheste **A**nfangszeit erster Aufgabe = 1
  - ⇨ früheste **A**nfangszeit von Folgeaufgabe = Maximum(früheste Anfangszeit + Dauer einer Vorgängeraufgabe)
- **Rückwärtsberechnung** spätester Endzeiten für Aufgaben:
  - ⇨ späteste **E**ndzeit letzter Aufgabe = früheste Anfangszeit + Dauer - 1
  - ⇨ späteste **E**ndzeit von Vorgängeraufgabe = Minimum(späteste Endzeit - Dauer einer Nachfolgeraufgabe)
- **kritische Aufgabe** auf kritischem Pfad (Anfangs- und Endzeiten liegen fest):
  - ⇨ früheste Anfangszeit = späteste Anfangszeit := späteste Endzeit - Dauer +1
  - ⇨ späteste Endzeit = früheste Endzeit := früheste Anfangszeit + Dauer - 1
- **kritische Kante** auf kritischem Pfad (zwischen zwei kritischen Aufgaben):
  - ⇨ früheste Endzeit Kantenquelle + 1 = späteste Anfangszeit Kantensenke
- **Pufferzeit** nichtkritischer Aufgabe (Zeit um die Beginn verschoben werden kann):
  - ⇨ Pufferzeit = späteste Anfangszeit - früheste Anfangszeit = ...

Probleme mit der Planung des Beispiels:

- zu viele Aufgaben liegen auf kritischen Pfaden (wenn kritische Aufgabe länger als geschätzt dauert, schlägt das auf Gesamtprojektlaufzeit durch)
  - ⇨ an einigen Stellen zusätzliche Pufferzeiten einplanen (um Krankheiten Fehlschätzungen, … auffangen zu können)
- einige eigentlich parallel ausführbare Aufgaben sollen von der selben Person bearbeitet werden
  - ⇨ Ressourcenplanung für Aufgaben durchführen
  - ⇨ Aufgaben serialisieren um „Ressourcenkonflikte“ aufzulösen
  - ⇨ (oder: weitere Personen im Projekt beschäftigen)
- Umrechnung auf konkrete Datumsangaben fehlt noch
  - ⇨ Berücksichtigung von Wochenenden (5 Arbeitstage pro Woche)
  - ⇨ ggf. auch Berücksichtigung von Urlaubszeiten

Verfeinerung von Abhängigkeiten und Zeitplanung:

Bislang gilt für abhängige Aktivitäten A1 und A2 (mit A2 hängt von A1 ab):

- **„Finish-to-Start“-Folge (FS) = Normalfolge**:
  Nachfolgeaktivität A2 kann erst bearbeitet werden, wenn Vorgänger A1 vollständig abgeschlossen ist: A1.Finish+1+lag ≤ A2.Start

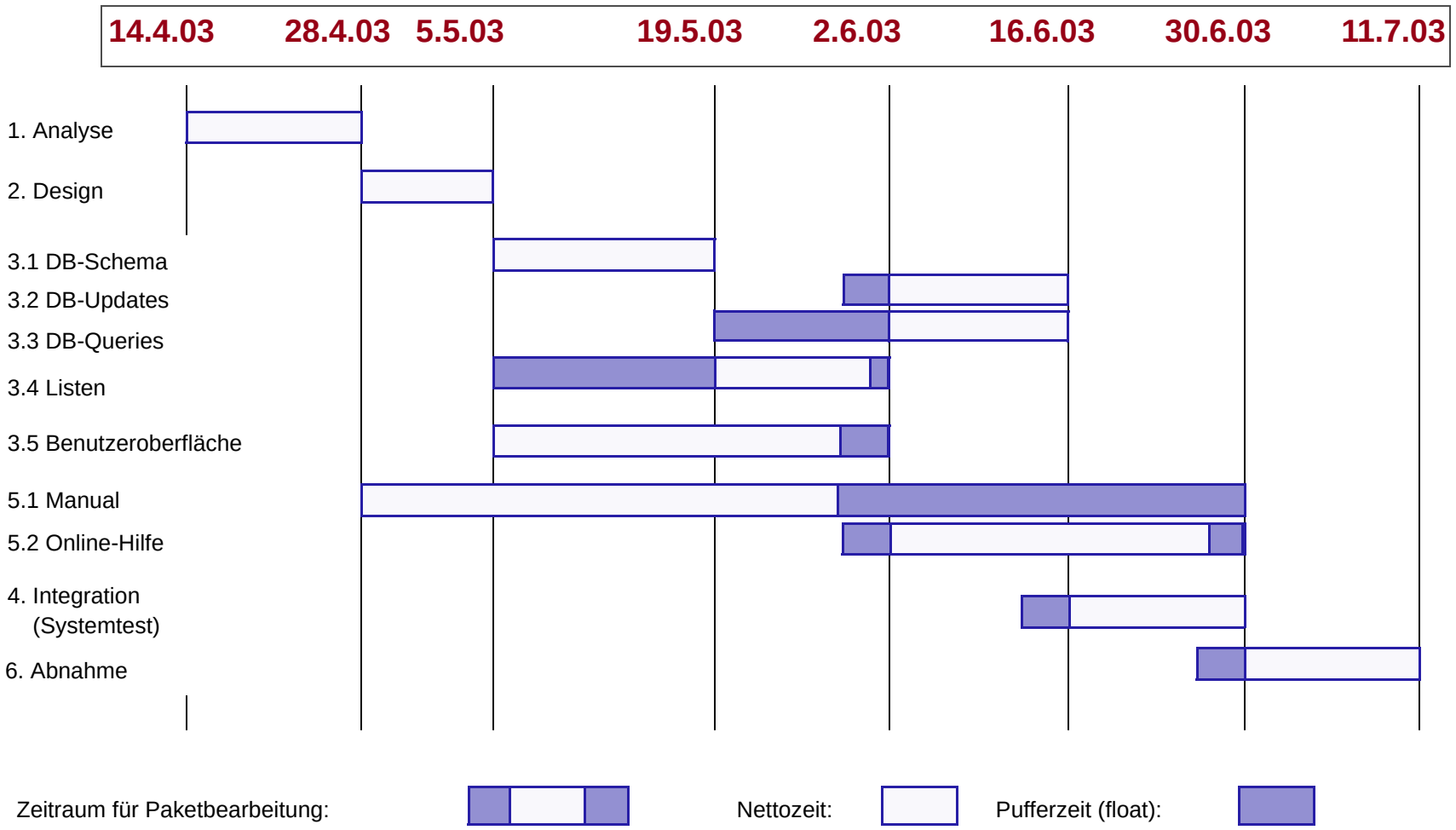
Manchmal lassen sich aber abhängige Aktivitäten auch parallel bearbeiten:

- **„Start-to-Start“-Folge (SS) = Anfangsfolge**:
  Aktivitäten können gleichzeitig beginnen, also: A1.Start+lag ≤ A2.Start
- **„Finish-to-Finish“-Folge (FF) = Endfolge**:
  Aktivitäten können gleichzeitig enden, also: A1.Finish+lag ≤ A2.Finish
- **„Start-to-Finish“-Folge (SF) = Sprungfolge (nicht benutzen)**:
  A1.Start+lag ≤ A2.Finish (früher für Rückwärtsrechnungen eingesetzt)

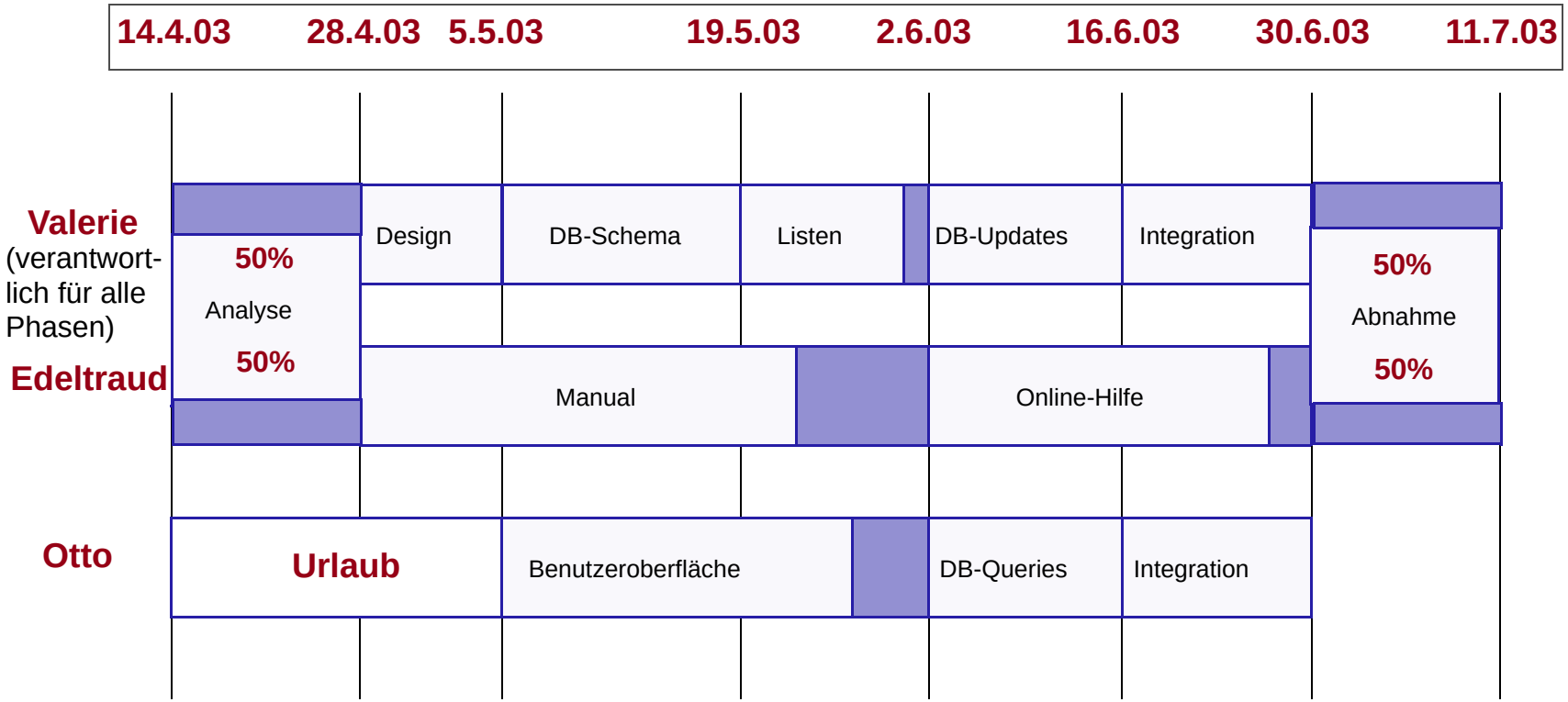
Des weiteren sind manchmal zusätzliche Verzögerungszeiten sinnvoll:

- frühestmöglicher Beginn einer Aktivität wird um **Verzögerungszeit (lag)** nach vorne oder hinten verschoben (für feste Puffer, Überlappungen)

Balkendiagramm (Gantt Chart, 1917 von Henry Gantt erfunden) - Idee:



Balkendiagramm mit Personalplanung:



Nettozeit:

Pufferzeit:
(float)

Achtung:

Durch Ressourcenzuteilung entstehen zusätzliche Zeit-restriktionen (z.B. Listen nach DB-Schema)

5.6 Weitere Literatur

[ACF97] V. Ambriola, R. Conradi, A. Fugetta: *Assessing Process-Centered Software Engineering Environments*, ACM TOSEM, Vol. 6, No. 3, ACM Press (1997), S. 283-328

Nicht zu alter Aufsatz mit Überblickscharakter zum Thema dieses Kapitels. Beschränkt sich allerdings im wesentlichen darauf, die drei Systeme OIKOS (Ambriola), EPOS (Conradi) und SPADE (Fugetta) der drei Autoren miteinander zu vergleichen. Es handelt sich dabei um Systeme der zweiten Generation, die in diesem Kapitel nicht vorgestellt wurden.

[Ba98] H. Balzert: *Lehrbuch der Softwaretechnik (Band 2): Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Spektrum Akademischer Verlag (1998)

Hier findet man (fast) alles Wissenswerte zum Thema Management der Software-Entwicklung.

[BP84] V.R. Basili, B.T. Perricone: *Software Errors and Complexity: An Empirical Investigation*, Communications of the ACM, Vol. 27, No. 1, 42-52, ACM Press (1984)

Eine der ersten Publikationen zu empirischen Untersuchungen über den Zusammenhang von Softwarekomplexität und Fehlerhäufigkeit

[Be99] K. Beck: *Extreme Programming Explained - Embrace the Change*, Addison Wesley (1999)

Eins der Standardwerke zum Thema XP, geschrieben vom Erfinder. Mehr Bettlektüre mit Hintergrundinformationen und Motivation für XP-Techniken, als konkrete Handlungsanweisung.

[Dy02] K.M. Dymond: *CMM Handbuch*, Springer Verlag (2002)

Einfach zu verstehende schrittweise Einführung in CMM (Levels und Upgrades von einem Level zum nächsten) in deutscher Sprache.

[Hu99] R. Hürten: Function-Point-Analysis - Theorie und Praxis (Die Grundlage für ein modernes Software-Management), expert-verlag (1999), 177 Seiten

Kompaktes Buch zur Kostenschätzung mit Function-Point-Methode, das Wert auf nachvollziehbare Beispiele legt. Zusätzlich gibt es eine Floppy-Disc mit entsprechenden Excel-Sheets.

[Hu96] W.S. Humphrey: *Introduction to the Personal Software Process*, SEI Series in Software Engineering, Addison Wesley (1996)

Das CMM-Modell für den „kleinen Mann". Das Buch beschreibt wie man als einzelner Softwareentwickler von bescheidenen Anfängen ausgehend die Prinzipien der Prozessüberwachung und -verbesserung einsetzen kann. Zielsetzungen sind zuverlässigere Zeit- und Kostenschätzungen, Fehlerreduktion, … .

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*, Addison Wesley (1999)

Präsentation des auf die UML zugeschnittenen Vorgehensmodells der Softwareentwicklung, eine Variante des hier vorgestellten Rational Unified (Objectory) Prozesses.

[Jo92] C. Jones: *CASE's Missing Elements*, IEEE Spektrum, Juni 1992, S. 38-41, IEEE Computer Society Press (1992)

Enthält u.a. ein vielzitiertes Diagramm über Produktivitäts- und Qualitätsverlauf bei Einsatz von CASE.

[OHJ99] B. Oestereich (Hrsg.), P. Hruschka, N. Josuttis, H. Kocher, H. Krasemann, M. Reinhold: *Erfolgreich mit Objektorientierung: Vorgehensmodelle und Managementpraktiken für die objektorientierte Softwareentwicklung*, Oldenbourg Verlag (1999)

Ein von Praktikern geschriebenes Buch mit einer Fülle von Tipps und Tricks. Es enthält eine kurze Einführung in den "Unified Software Development Process", sowie in das V-Modell.